Modelling and Automatic Enforcement of Architectural Design Rules

AUTHOR
Anders Jörgen Mattsson

A THESIS SUBMITTED FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
AT UNIVERSITY OF LIMERICK, IRELAND

Supervised by:
Prof. Brian Fitzgerald
Dr. Björn Lundell

SUBMITTED TO THE UNIVERSITY OF LIMERICK May 2012
DECLARATION

May, 2012

The substance of this thesis is the original work of the author and due reference and acknowledgement has been made, when necessary, to the work of others. No part of this thesis has been accepted for any degree and is not being concurrently submitted for any other award.

Anders Mattsson
(Candidate)
Table of Contents

ABSTRACT ................................................................................................................................. 1

ACKNOWLEDGEMENTS ........................................................................................................... 3

LIST OF FIGURES ....................................................................................................................... 5

LIST OF TABLES .......................................................................................................................... 7

PRESENTATION OF THESIS ....................................................................................................... 9

PART I INTRODUCTION ............................................................................................................. 11

1 Problem Overview ................................................................................................................... 13
   1.1 Model Driven Development ............................................................................................ 14
   1.2 Software Architecture .................................................................................................... 22
2 Scope of Current Work ............................................................................................................ 30
3 Research Method .................................................................................................................... 32
   3.1 Selection of Research Methods ...................................................................................... 32
   3.2 Conduction of the Research .......................................................................................... 37
   3.3 On the Validity of the Results ....................................................................................... 39
4 Contribution ........................................................................................................................... 40
   4.1 Contribution 1: An illustration of how insufficient solutions on how to model architectural design rules are a serious problem in projects using MDD. ......................................................... 42
   4.2 Contribution 2: A method, suitable for practice, with prototypic tool support, on how to model and automatically enforce architectural design rules. ................................................................................. 44
   4.3 Contribution 3: A validation of the method in an industrial development project. ........... 48
5 Summary and Discussion ........................................................................................................ 54
6 Organization of the rest of this Thesis .................................................................................. 58
References ..................................................................................................................................... 58

PART II THE PAPERS ................................................................................................................. 67

PAPER I ........................................................................................................................................ 69
   Abstract ................................................................................................................................... 70
   1 Introduction .......................................................................................................................... 71
   2 Modelling Architectural Design Rules and MDD ............................................................... 72
      2.1 The role of Architectural Design Rules ........................................................................ 72
      2.2 MDD and Architectural Design Rules ......................................................................... 75
      2.3 Modelling Architectural Design Rules ......................................................................... 77
   3 Research approach .............................................................................................................. 78
   4 Integrating Architectural Design Rules in a MDD project ................................................ 81
      4.1 Action Case Context ..................................................................................................... 81
      4.2 Project Challenges ........................................................................................................ 82
      4.3 Rationale for choosing MDD ...................................................................................... 83
      4.4 Tool selection ............................................................................................................... 84
   5 The process .......................................................................................................................... 84
Table of Contents

5.1 Capturing the architecture ......................................................................................... 85
5.2 Manual Reviews of Architectural Conformance .......................................................... 90
6 Conclusions .................................................................................................................. 91
Acknowledgments ............................................................................................................ 92
References ....................................................................................................................... 92

PAPER II ............................................................................................................................. 97

Abstract ............................................................................................................................. 98
1 Introduction .................................................................................................................... 99
2 Background .................................................................................................................. 100
  2.1 Architectural Design Rules and MDD ........................................................................ 100
  2.2 Architectural Design Rules in an MDD Context Using UML ........................................ 104
3 An Illustrative Example .................................................................................................. 107
  3.1 Traditional Way of Modelling the Architecture ............................................................ 109
  3.2 Modelling the Architecture According to Our Approach ............................................. 111
4 Research Approach ...................................................................................................... 114
5 An Approach to Modelling Architectural Design Rules .................................................. 116
  5.1 General Transformations ........................................................................................... 116
  5.2 Additional UML Specific Transformations ................................................................. 120
6 Automating the Approach ............................................................................................ 126
7 Modelling Architectural Design Rules of an Industrial Strength System ....................... 128
8 Discussion and Conclusions ......................................................................................... 132
Appendixes ...................................................................................................................... 134
  Appendix A. Formal definition of transformations ......................................................... 134
References ....................................................................................................................... 154

PAPER III ............................................................................................................................ 159

Abstract ............................................................................................................................. 160
1 Introduction .................................................................................................................... 161
2 On the investigated approach ....................................................................................... 163
3 Research approach ....................................................................................................... 165
  3.1 On the research approach adopted .......................................................................... 165
  3.2 Data collection and analysis .................................................................................... 167
4 Findings ........................................................................................................................ 168
  4.1 Action Case Context ................................................................................................. 168
  4.2 Findings in development artifacts .......................................................................... 169
  4.3 Effects on productivity ......................................................................................... 170
  4.4 Effects on quality .................................................................................................... 171
  4.5 Learnability of the method .................................................................................... 172
5 Conclusions .................................................................................................................. 173
References ....................................................................................................................... 175
Appendix I ......................................................................................................................... 178

BIBLIOGRAPHY AND REFERENCES ................................................................................... 183
Abstract

A basic premise of Model Driven Development (MDD) is to capture all important design information in a set of formal or semi-formal models which are then automatically kept consistent by tools. The concept however is still relatively immature and there is a lack of empirically validated guidelines. This thesis reports on the use of MDD in significant real-world projects over several years. Our research has found current techniques for modelling software architecture insufficient for modelling of architectural design rules. As a result developers have to rely on time-consuming and error-prone manual practices to keep a system consistent with its architecture. This thesis addresses this problem by presenting an approach that enables automatic enforcement of the architectural rules on the detailed design. The approach consists of a method on how to model architectural design rules with associated automation that is demonstrated through a prototype tool which upon request checks that the system model conforms to the architectural rules and report all violations. The rules are modelled in a subset of UML at the abstraction level of the meaning of the rules. The high abstraction level and the use of UML make the rules both amenable to automation and easy to understand for both architects and developers, which is crucial to deployment in an organization. Results from an action case study on an industrial development project indicate a high Return On Investment (ROI) for using the approach since it yields a substantial increase in both productivity and quality, while the actual investment required for training and tooling is small. In addition the approach is very well received by the users who report increased satisfaction in their work.
Acknowledgements

The work presented in this thesis would not have been possible without the support and influence of a number of people who I have had the fortune and privilege to meet.

First of all I would like to thank my supervisors Dr. Björn Lundell and Professor Brian Fitzgerald for their guidance and support throughout the whole project. A special thank you goes to Dr. Brian Lings who has acted as an extra advisor and given me much invaluable advice along the road.

During this research I have had the opportunity to work in two pan-European research projects within the Eureka initiative: COSI and MoSiS. During these projects I have worked together with many interesting and knowledgeable people coming from many different countries. This has been extremely inspiring, thank you all for that!

This work would not have been possible without the support from Combitech, Saab, Vinnova and Lero providing me with resources for my research. I specially want to thank the former president for Combitech, Christer Hoberg for giving me the opportunity and my manager Anders Åström for supporting me during the project. Also, many thanks to all colleagues at Combitech, Lero and University of Skövde that have been involved in one way or another.

Finally and most importantly I want to thank my wife Maria and our children Matilda, Filip and Linnea for their understanding and support during this sometimes rather introvert journey, I love you all!

Anders Mattsson
Ulricehamn, April 2011
List of Figures

Introduction

Figure 1  An overview of MDA 17
Figure 2  Domain Specific Modelling 19
Figure 3  Definition of the architectural design rule example using OCL 29
Figure 4  A framework for Integrating Research Perspectives 34
Figure 5  Research process. 37
Figure 6  The phases of the project 43
Figure 7  An approach for automatic enforcement of architectural design rules. 46
Figure 8  The rules in Figure 3 modelled according to the approach presented in this thesis 46
Figure 9  The rule in Figure 8 modelled in GOPRR 47
Figure 10 Changes to the development process 50

Paper I

Fig. 1  An overview of MDA 75
Fig. 2  Domain Specific Modelling 77
Fig. 3  A framework for Integrating Research Perspectives 79
Fig. 4  The phases of the project 85
Fig. 5  The layers of the system 86
Fig. 6  Executable framework of the architecture 88

Paper II

Fig. 1  Defining a stereotype Data_Class in a UML profile 105
Fig. 2  Defining a class Position of the stereotype Data_Class in a UML model 105
Fig. 3  Definition of the architectural design rule example using OCL 106
Fig. 4  Definition of the architectural design rule example using specialisation of the UML meta-model 106
Fig. 5  Capturing the architectural design rule example using the approach suggested in [Fuentes-Fernández and Vallecillo-Moreno 2004] 107
Fig. 6  High-level structure and framework in UML using the traditional way of modelling architecture 110
Fig. 7  Architectural design rules for Data_Items modelled according to our approach 112
Fig. 8  High-level structure of system model in UML using our approach 113
Fig. 9  Part of the detailed design for the Data_Items package in the system model 113
Fig. 10 A generic architectural rules model used in the definition of the transformations 117
### List of Figures

<table>
<thead>
<tr>
<th>Fig.</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>Example of transformation according to transformation T1.</td>
<td>117</td>
</tr>
<tr>
<td>12</td>
<td>An example of using transformation T2, T3 and T4.</td>
<td>118</td>
</tr>
<tr>
<td>13</td>
<td>An example of using transformation T7</td>
<td>119</td>
</tr>
<tr>
<td>14</td>
<td>Capturing an association constraint in a meta-model using the general transformations.</td>
<td>120</td>
</tr>
<tr>
<td>15</td>
<td>A generic architectural rules model used in the definitions of the transformations</td>
<td>121</td>
</tr>
<tr>
<td>16</td>
<td>Example of rules on package containment.</td>
<td>121</td>
</tr>
<tr>
<td>17</td>
<td>Example of rules on associations</td>
<td>122</td>
</tr>
<tr>
<td>18</td>
<td>Example of rules on generalizations, dependencies and realizations</td>
<td>123</td>
</tr>
<tr>
<td>19</td>
<td>Example of rules on attributes.</td>
<td>123</td>
</tr>
<tr>
<td>20</td>
<td>Example of rules on operations</td>
<td>124</td>
</tr>
<tr>
<td>20</td>
<td>Example of rules on operations</td>
<td>124</td>
</tr>
<tr>
<td>21</td>
<td>Example on rules on state machines</td>
<td>125</td>
</tr>
<tr>
<td>22</td>
<td>Defining a stereotype Data_Class in a UML profile</td>
<td>126</td>
</tr>
<tr>
<td>23</td>
<td>Screenshot of tool</td>
<td>128</td>
</tr>
<tr>
<td>24</td>
<td>Part of the architectural rules model</td>
<td>131</td>
</tr>
<tr>
<td>25</td>
<td>Top level of system model</td>
<td>131</td>
</tr>
<tr>
<td>26</td>
<td>The ClockDevice in the system model</td>
<td>132</td>
</tr>
</tbody>
</table>

### Paper III

<table>
<thead>
<tr>
<th>Fig.</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>An approach for automatic enforcement of architectural design rules.</td>
<td>164</td>
</tr>
<tr>
<td>2</td>
<td>An architectural rule modeled according to the novel approach</td>
<td>164</td>
</tr>
<tr>
<td>3</td>
<td>Model of rules applying to components in the logical view</td>
<td>170</td>
</tr>
<tr>
<td>4</td>
<td>A generic architectural rules model used in the definition of the transformations</td>
<td>179</td>
</tr>
</tbody>
</table>
List of Tables

Introduction
Table 1  Contribution, Research Methods and Papers  40
Table 2  A subset of the original architectural design rules  48

Paper II
Table 1  A subset of the full table which had mappings between all the original architectural design rules and resulting modelling constructs  129
Table 2  Usage frequency and violation percentage for each transformation  130

Paper III
Table 1  Definition of transformations between constructs in the architectural rules model and constraints on stereotypes in the system model  179
Presentation of Thesis

This thesis has been prepared in the style “Thesis by Publication.” Therefore referencing styles vary throughout the thesis as required by the specific publication. Table 1 below outlines the papers in part II of the thesis and the publications in which that paper has been published or accepted for publication.

Table 1. Publication for each paper of the thesis.

<table>
<thead>
<tr>
<th>Paper</th>
<th>Publication</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>IEEE Transactions on Software Engineering</td>
</tr>
<tr>
<td>II</td>
<td>ACM Transactions on Software Engineering and Methodology</td>
</tr>
<tr>
<td>III</td>
<td>Proceedings of Informing Science &amp; IT Education Conference (InSITE) 2012</td>
</tr>
</tbody>
</table>
Part I

Introduction
1 Problem Overview

Model-Driven Development (MDD) (Selic 2003) is an emerging discipline (Schmidt 2006, Balasubramanian et al. 2006) where the prevalent software-development practices in industry are still immature (Hailpern and Tarr 2006). The success of MDD in practice is currently an open question (Hailpern and Tarr 2006) and there is a lack of proven real-world usage of MDD in large industrial projects.

A basic premise of MDD is to capture all important design information in a set of formal or semi-formal models that are automatically kept consistent by tools. The purpose is to raise the level of abstraction at which the developers work and to eliminate time-consuming and error-prone manual work in maintaining consistency between different design artefacts, such as UML diagrams and code (Hailpern and Tarr 2006). An important design artefact in any software development project, with the possible exception of very small projects, is the software architecture (Bass et al. 2003). An important part of any architecture is the set of architectural design rules. This thesis defines architectural design rules as the rules, specified by the architect(s), which need to be followed in the detailed design of the system. The importance of architectural design rules has been further highlighted by recent research on software architecture (Kruchten 2004a, Jansen and Bosch 2005, Tyree and Akerman 2005, Jansen et al. 2007, Kruchten et al. 2006a) which has acknowledged that a primary role of the architecture is to capture the architectural design decisions. An important part of these design decisions consists of architectural design rules. In a MDD context the design of the system is captured in models of the system, therefore architectural design rules in a MDD context specify rules that the models of the system have to conform to. There is however no satisfactory solution in the existing body of literature on how to model architectural design rules.

The inability to formalize the architectural design rules leads to a need for manual enforcement of them. The research presented in this thesis shows that this is an error-prone and time-consuming task that takes most of the effort of the architects during the construction-intensive phases of a project. This problem exists in traditional document-based development as well as in MDD but it is more apparent and acute in MDD. This is because MDD has been able to automate the step from detailed design to
implementation eliminating time consuming coding and code reviews. However, MDD has not been able to automate enforcement of the architecture on the detailed design due to the inability to model architectural design rules. The case presented in Paper I shows that the inability to model architectural design rules makes architectural enforcement a bottleneck in MDD projects.

In the following two sections Model Driven Development and Software Architecture is discussed as a background to the research presented in this thesis.

1.1 Model Driven Development

1.1.1 Background

Ever since the early days of computing there has been a trend to raise the level of abstraction on which software is defined. The intent has been to make software development more efficient by allowing programming in terms of design intent rather than in terms of the underlying computing hardware such as CPUs, memory and input/output devices and by shielding the developers from the complexity in the underlying technology. These abstractions have included both languages and platforms. As an example early languages such as assembly and Fortran shielded the programmers from the complexities of machine code and early operating systems such as OS/360 and Unix shielded the programmers from the complexities in the hardware. The trend has gone from machine code (1\textsuperscript{st} generation) through assembly language (2\textsuperscript{nd} generation) to Fortran, C, C++ (3\textsuperscript{rd} generation) and other current programming language. MDD can be seen as a natural continuation of this trend where programs are defined with graphical models instead of using text files.

In addition to producing software code, software engineering methodologies prescribe that various other information such as requirements, architecture, design and testing information are produced and maintained during development. A basic idea of MDD is to ease the production and maintenance of this information by capturing it in models and the use of tools to automatically keep it consistent. Thus, the purpose of MDD is to raise the level of abstraction at which the developers work and to eliminate time consuming and error prone manual work in keeping different design artefacts consistent (Hailpern and Tarr 2006).
Introduction

Using graphical models has a long tradition in software engineering where for example Goldstine and von Neumann’s program flowchart technique was developed in the 1940’s. State charts, sequence diagrams, and class diagrams are just a small sample of diagrams that have been in frequent use for several decades. To be useful and effective a software engineering model must however posses certain characteristics. According to Selic (2003) five such key characteristics are abstraction, understandability, accuracy, predictiveness and inexpensive, each discussed below:

- **Abstraction**: A model is always a reduced rendering of the system it represents. By removing detail that is irrelevant for a given viewpoint, it lets us understand the essence more easily.

- **Understandability**: In order to be understandable it is not sufficient to just abstract away details. The model must also be presented in a form that most directly appeals to our intuition. Graphical or visual representation of models are in many cases more effective than textual representation in communicating information to humans because they tap into the capabilities of the powerful and highly parallel human visual system. We like receiving information in visual form and can process it very efficiently (Moody 2009). Also, graphical notations are two dimensional (spatial) while textual notations are one-dimensional (linear). This allows direct representation of graphs such as state charts, flow charts and class diagrams in a graphical notation while these can only be indirectly represented in text, which requires the reader of the text to reengineer the graph, on paper or in mind, from the text. Selic argues that the amount of information that must be absorbed and recognized to understand linear programs is enormous and requires significant intellectual effort.

- **Accuracy**: A model must provide a true-to-life representation of the modeled system’s features of interest. This requires that the language used to describe the model has formal semantics, that is, that models expressed in the language only can be interpreted in one well defined way.

- **Predictiveness**: To be useful it should be possible use a model to correctly predict interesting but non obvious system properties. Clearly this depends greatly on the model’s accuracy.

- **Inexpensive**: It must be significantly cheaper to construct and analyze the model than the modeled system.
MDD is not the first attempt to use models to elevate the level of abstraction of software design and keeping design artifacts in sync. In the 1980s much effort was spent on what was called Computer-Aided Software Engineering (CASE). This effort focused on developing tool support for the various methods for structured analysis and design (SA/SD) that emerged during the time. These methods are based on functional decomposition where the system is seen as a process transforming input data flows into output data flows. This process is then recursively decomposed into smaller subprocesses connected by data flows creating a hierarchy of processes connected with data flows. CASE suffered from several problems. One major problem was an inability to scale to handle complex production scale systems due to both limitations in tool support for multi user support (Schmidt 2006) and problems in handling changeability, independent parallel development and comprehensibility with functional decomposition (Parnas 1972). Another major problem was that there were no direct relation between the models and implementation which often resulted in that these drifted apart, making the models more or less useless during the later phases of the project.

In the early part of the 1990s the object paradigm and related technologies gained a heightened interest, boosted by expectations to better handle the problems of functional decomposition pointed out by Parnas (1972). Object-Oriented programming languages such as C++, Java and Smalltalk became widely used. These were followed by a multitude of Object-Oriented methods such as Booch, Coad-Yourdon, OMT, Shlaer-Mellor, OOSE and over forty others that appeared during the period between 1989 and 1994, a period called “the method wars” (Booch et al. 1999). There was a great deal of overlap and conceptual alignment across the methods, much of it obscured by notational and other differences of little or no consequence. This led to much confusion and unnecessary fragmentation causing an impediment to adoption of the methods. As a response to this situation Grady Booch and James Rumbaugh, the developers of two of the most widespread methods, the Booch method and the OMT method, joined forces to unify the methods into what was called the Unified Method in October 1994. At the end of 1995 Ivar Jacobsson, the creator of the OOSE method, joined the effort and the scope of the effort was expanded to incorporate OOSE. At this time the scope of the effort had also changed to only define a unified modelling language, not a whole method, and was subsequently renamed to Unified Modelling Language, UML. The first release of UML was version 0.9 in June 1996. At this time a consortium consisting of several major
software companies was formed to work on the 1.0 version of UML. This was submitted to the OMG (Object Management Group) as a proposal for a standard for object oriented modeling. It was accepted in November 1997. Since then the OMG has had the responsibility for developing the standard (Kruchten et al. 2006a). At the time of writing the current version is 2.3, released in May 2010.

Today there exist several approaches to Model-Driven Development (MDD), all building on the object paradigm, such as OMG’s Model-Driven Architecture (MDA) (OMG 2003a), Domain Specific Modelling (DSM) (Karsai et al. 2003, Tolvanen and Kelly 2005), and Software Factories (Greenfield and Short 2004) from Microsoft. Each of these is discussed in separate sub-sections below.

1.1.2 Model-Driven Architecture (MDA)

MDA prescribes that three models, or sets of models, shall be developed as illustrated in Figure 1:

![Figure 1. An overview of MDA](image)

1. The Computational Independent Model(s) (CIM) captures the requirements of the system.
2. The Platform Independent Model(s) (PIM) captures the system’s functionality without considering any particular execution platform.
3. The Platform Specific Model(s) (PSM) combines the specifications in the PIM with the details that specify how the system uses a particular type of platform.
The PSM is a transformation of the PIM using a mapping either on the type level or at the instance level. A type level mapping maps types of the PIM language to types of the PSM language. An instance level mapping uses marks that represent concepts in the PSM (such as a process or a CORBA object). When a PIM shall be deployed on a certain platform the marks are applied to the elements of the PIM before the transformation.

MDA does not prescribe any particular language to be used for the models, but UML (OMG 2007) is proposed as one possibility. There is also an accompanying OMG standard, MOF (Meta-Object Facility) (OMG 2006), which can be used to describe meta-models for modelling languages (meta-models are further discussed in the next sub-section).

1.1.3 Domain Specific Modelling (DSM)

Another approach to MDD is Domain Specific Modelling (DSM). The basic idea of Domain Specific Modelling (DSM) is that instead of using a general purpose language such as UML to model the system one define and use a language that is specifically well suited to define systems in a narrow domain. In Figure 2 the basic approach of DSM is illustrated. A Domain Specific Modelling Language (DSML) is defined that captures the main concepts in the domain of an application. This DSML definition is then used as input into a language configurable modelling tool in which the system is modelled in the DSML. Examples of this approach are described in Karsai et al. (2003) and Tolvanen and Kelly (2005).
The DSML is defined by an abstract syntax, a concrete syntax, semantics, a mapping between the abstract syntax and the concrete syntax and a mapping between the abstract syntax and the semantics. The abstract syntax is typically defined in a model that defines the concepts of the domain, relationships and integrity constraints. For this one can, for instance, use MOF (OMG 2006) and OCL (OMG 2003b). Since this is a model which in turn defines elements of a language for a model it is called a meta-model (Atkinson and Kuhne 2003). The concrete syntax of the language comprises the visual symbols that represent the concepts in the meta-model. To provide this and the mapping to the abstract syntax one must specify symbols for the model elements in the meta-model. The semantics of the language define the meaning of the concepts in the abstract syntax. Defining semantics and mapping these to the abstract syntax can be done by providing transformations of the meta-model into another language with defined semantics, such as a programming language. To define such transformations one can for instance use the OMG QVT (Query/View/Transformation) standard (OMG 2011). DSM can be seen as a special case of MDA where a domain specific language is defined for use in the PIM.

1.1.4 Software Factories

Software factories is a MDD approach that incorporates the DSM idea but goes further since it provides a method for building complete customized tool chains for product families using extensible tools. Such a tool chain is called a software factory.
A software factory provides a production facility for a product family by configuring extensible tools such as Eclipse or Microsoft Visual Team System using a software factory template based on a software factory schema. To leverage models effectively for various forms of automation, software factories make heavy use of domain-specific languages (DSLs).

A software factory schema is a document that defines a number of viewpoints relevant to stakeholders in the production of the target system. Each viewpoint defines the artifacts produced and consumed by its stakeholders, the activities they perform against those artifacts, and the reusable assets available to support them in performing those activities. Examples of artifacts are XML documents, models, configuration files, build scripts, source code files, SQL files, localization files, deployment manifests and test case definitions. The schema also defines relationships or mappings between the artifacts, so that consistency among them can be maintained.

A software factory template is the implementation of a software factory schema. A software factory template includes code and metadata that can be loaded into extensible tools, like an Interactive Development Environment (IDE), or an enterprise life cycle tool suite, to automate the development and maintenance of family members. It is called a software factory template because it configures the tools to produce a specific type of software, just as a document template loaded into a tool like Microsoft Word or Excel configures it to produce a specific type of document.

1.1.5 Problems in MDD

Even though MDD shows great potential it is still an emerging discipline where there are many problems. Selic (2003) argues that failure to meet one or more of the previously listed characteristics for useful and effective models may be the main reason for the limited success for modelling techniques. In agreement with Selic these characteristics are used to group problems and issues raised by MDD.

Abstraction and understandability: As pointed out by Hailpern and Tarr (2006) there is a risk that MDD moves complexity rather than reducing it, from artifacts to artifact relationships and tools. Each type of model requires a particular set of skills to produce and evolve effectively. On one hand, in raising the level of developer abstraction to
models, MDD enables specialists to work with abstractions that better suit their tasks and expertise. On the other hand they must understand how a change to their artifacts relates to or affects other related artifacts that could be described in different notations from the ones they are familiar with. This may mean that they have to understand a wide set of notations as well as being fluent in various transformation notations and transformations may be extremely complex.

**Accuracy and predictiveness:** Selic (2003) argues that many modelling techniques are weak in terms of accuracy, which also means that the models are weak in predictability. Also, if the models are not formally connected to the software code there is no way to ensure that the programmers follows the design decisions captured in the models or to keep the models in sync with the software code, leading to a successive digression between models and software code. This opinion is supported by Hailpern and Tarr (2006) that points out that failure to automatically keep artifacts, including code consistent are significant problems for MDD.

The standardization of modelling notations such as UML is unquestionably an important step for achieving MDD. Standardization provides developers with uniform modelling notations for a wide range of modelling activities. Standardization efforts also enables many types of tooling support for creating and manipulating models in novel ways, generating artifacts (such as code) from models, and reverse engineering models from other artifacts. Unfortunately, UML, the most widespread standardized MDD language has been noted to have some serious problems. It has become enormous and unwieldy in an attempt to address too many needs. It lacks semantic formalism in many parts such as use cases, that is, it violates the Accuracy characteristic for a model. This lack of semantics complicates accurate usage of UML and limits the ability of tool vendors to provide reliable, consistent model technologies.

**Inexpensive:** Current coding tools have productivity features that still are missing in modelling tools, such as code completion and refactoring support. This is an impediment to acceptance of MDD for many developers who feel that being forced to use MDD tools make them less productive.
1.2 Software Architecture

1.2.1 Background

Important precursors to Software Architecture during the 1970s and the 1980s were the work on “Information Hiding” (Parnas 1972, Parnas 1976, Parnas et al. 1985) and “abstract data types” (Liskov and Zilles 1974). This work claimed that in order to be resistant to change the software should be decomposed into modules containing both functionality and data only accessible by operations provided by the module.

Software Architecture as a distinct discipline started to emerge in the early 1990s. In 1992 Perry and Wolf published their seminal article “Foundations for the study of Software Architecture” (Perry and Wolf 1992), perhaps most famous for its simple formula: Software Architecture = {Elements, Form, Rationale}. In this formula ‘Elements’ are either processing, data or connecting elements. ‘Form’ represents constraints on these elements and ‘Rationale’ is the motivation for the architecture.

Elements have subsequently been separated into components and connectors by many researchers. In 1996 the seminal book “Software architecture: perspectives on an emerging discipline” by Shaw and Garlan (1996) was published which firmly established Software Architecture as a distinct discipline separated from software design and programming based on the view of software architecture as a set of components and connectors.

1.2.2 Software architecture as a set of components and connectors

From mid 1990s and forward a number of architectural description languages (ADL) (Medvidovic et al. 2007, Medvidovic and Taylor 2000) were defined. Examples of languages are AADL, ACME, Aesop, C2, Darwin, Koala, MetaH, Rapide, SADL, UML, UniCon, Weaves and Wright. These facilitate the description of an architecture’s components and connections. The languages are usually graphical, and provide some form of "box and line" syntax for specifying components and hooking them together with connectors.

During the same time several methods for creating and analyzing an architecture have been developed, such as:
Introduction

- Attribute-Driven Design (ADD) (Wojcik et al. 2006, Bass et al. 2003) and Architecture Tradeoff Analysis Method (ATAM) (Kazman et al. 2000, Bass et al. 2003) developed at the CMU/SEI;
- The Rational Unified Process® (RUP) 4 + 1 views (Kruchten 1995, Kruchten 2004b) developed and commercialized by Rational Software, now IBM;
- The QASAR method (Bosch and Molin 1999, Bosch 2000, Bengtsson and Bosch 1998) developed by the RISE research group at the University of Karlskrona/Ronneby;
- Siemens’ 4 Views (S4V) method (Hofmeister et al. 2000, Soni et al. 1995), developed at Siemens Corporate Research;
- Business Architecture Process and Organization (BAPO/CAFCR) developed primarily at Philips Research (America et al. 2004, van der Linden et al. 2004);
- Architectural Separation of Concerns (ASC) (Ran 2000) developed at Nokia Research;
- Quality-driven Architecture Design and Analysis (QADA) (Matinlassi et al. 2002) developed at VTT.

There is also an IEEE standard (IEEE 2000) for describing the architecture of a software-intensive system. The architectural description is organized into a number of views where each view consists of a number of models. A view conforms to a viewpoint. The viewpoint determines the languages to be used to describe the view, and any associated modelling methods or analysis techniques to be applied to the models of the view. The standard does not prescribe which viewpoints or views to use and therefore not how models shall be documented.

Even though there is no generally agreed upon definition of Software Architecture (Kruchten et al. 2006b) there is a general agreement among researchers that an important purpose of the architecture is to guide and control the design of the system so that it meets its quality requirements, with quality requirements meaning requirements on non-functional properties such as availability, modifiability, performance, security, testability and usability. Bass et al. (2003) are unequivocal in stating the importance of an architectural approach to achieve quality requirements:
“The architecture serves as the blueprint for both the system and the project developing it. It defines the work assignments that must be carried out by design and implementation teams and it is the primary carrier of system qualities such as performance, modifiability, and security – none of which can be achieved without a unifying architectural vision. Architecture is an artefact for early analysis to make sure that the design approach will yield an acceptable system. Moreover, architecture holds the key to post-deployment system understanding, maintenance, and mining efforts. In short, architecture is the conceptual glue that holds every phase of the project together for all of its many stakeholders.”

A common understanding in architectural methods is that the architecture is represented as a set of components related to each other (Shaw et al. 1995, Perry and Wolf 1992). The components can be organized into different views focusing on different aspects of the system. Different methods propose different views; typical views are a view showing the development structure (e.g. packages and classes), a view showing the runtime structure (processes and objects) and a view showing the resource usage (processors and devices). In any view each component is specified with the following:

- An interface that documents how the component interacts with its environment.
- Constraints and rules that have to be fulfilled in the design of the component.
- Allocated functionality.
- Allocated requirements on quality attributes.

A typical method of decomposition (see for instance Bass et al. (2003), Wojcik et al. (2006), Bosch (2000) and Matinlassi et al. (2002)) is to select and combine a number of patterns that address the quality requirements of the system and use them to divide the functionality in the system into a number of elements. Child elements are recursively decomposed in the same way down to a level where no more decomposition is needed, as judged by the architect. The elements are then handed over to the designers who detail them to a level where they can be implemented. For common architectural patterns such as Model-View-Controller, Blackboard or Layers (Buschmann 1996) this typically means that you decompose your system into subsystems containing different kinds of classes (such as models, views and controllers). However the instantiation into actual classes is often left to the detailed design, for two main reasons:
1. Functionality will be added later, either because it was missed or because a new version of the system is developed, so more elements will be added later that also have to follow the design patterns decided by the architect.

2. It is not of architectural concern. The concern of the architect is that the design follows the selected architectural patterns, not to do the detailed design.

This means that a substantial part of the architecture consists of design rules on what kinds of elements, with behavioural and structural rules and constraints, there should be in a certain subsystem.

1.2.3 Software architecture as a set of design decisions

The predominant view of architecture as a set of components and connectors were challenged in the mid 2000s. In 2004 Jan Bosch suggested that instead of viewing software architecture as a set of components and connectors it should be viewed as the composition of a set of architectural design decisions (Bosch 2004). According to this, an architectural design decision consists of a solution part and a requirements part. The solution part is the first-class representation of a logically coherent structure that is imposed on the design decisions that have already been taken. The design decision may (1) add components to the architecture, (2) impose functionality on existing components, (3) add requirements on the expected behaviour of components and (4) add constraints or rules on part or all of the software architecture. This new line of research was motivated by Bosch by the following set of problems with the traditional view of the architecture as components and connectors (Bosch 2004):

- **Lack of first-class representation**: Architecture design decisions lack a first class representation in the software architecture. Once a number of design decisions are taken, the effect of individual decisions is implicitly present, but almost impossible to identify in the resulting architecture. Consequently, the knowledge about the “what and how” of the software architecture is quickly lost. Some architecture design methods stress the importance of documenting architecture design decisions, but experience shows that this documentation often is difficult to interpret and use by individuals not involved in the initial design of the system.
• **Design decisions cross-cutting and intertwined**: Architecture design decisions typically are cross-cutting the architecture, i.e. affect multiple components and connectors, and often become intimately intertwined with other design decisions.

• **High cost of change**: A consequent problem is that a software architecture, once implemented in the software system, is, sometimes prohibitively, expensive to change. Due to the lack of first-class representation and the intertwining with other design decisions, changing or removing existing design decisions is very difficult and affects many places in the system.

• **Design rules and constraints violated**: During the evolution of software systems, designers, and even architects, may easily violate the design rules and constraints imposed on the architecture by earlier design decisions.

• **Obsolete design decisions not removed**: Removing obsolete architecture design decisions from an implemented architecture is typically avoided, or performed only partially, because of the (1) effort required, (2) perceived lack of benefit and (3) concerns about the consequences, due to the lack of knowledge about the design decisions. The consequence, however, is the rapid erosion of the software system, resulting in high maintenance cost and, ultimately, the early retirement of the system.

This line of research has been followed by a number of researchers who have also widened the scope to architectural knowledge, emphasizing the importance of capturing rationale as a part of the design decisions (Kruchten et al. 2006a, Kruchten et al. 2009, Tyree and Akerman 2005).

1.2.4 Problems in Software Architecture

**No clear separation between software architecture and design.**

There is no satisfying, short, crisp answer to the question “what is software architecture?” – no widely accepted definition exists (Kruchten et al. 2006b, Falessi et al. 2009). Some wide spread definitions are:

• [Software architectures go] “*beyond the algorithms and data structures of the computation; designing and specifying the overall system structure emerges as a new kind of problem. Structural issues include gross organization and global control structure; protocols for communication, synchronization, and data access; assignment of functionality to design elements; physical distribution;*
composition of design elements; scaling and performance; and selection among design alternatives.” (Garlan and Shaw 1993)

- “The fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution.” (IEEE 2000)
- “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.” (Bass et al. 2003)

In particular there is no clear distinction between the terms architecture and design. Eden and Kazman note that:

“The terms “architecture”, “design” and “implementation” appear to connote varying degrees of abstraction in the continuum between complete details (“implementation”), few details (“design”), and the highest form of abstraction (“architecture”). ... A clear distinction has remained elusive” (Eden and Kazman 2003)

A general perception is that the architecture is a subset of the design, the “architectural significant” elements of the design as stated in the Rational Unified Process (Kruchten 2004b). Variations of this perception are for instance expressed by Fowler; “Architecture is about the important stuff” (Fowler 2003) and in Abrahamsson et al. (2010); “not all design is architecture. Architectural design decisions are a sub-set of the design decisions”.

This lack of separation is the cause of several problems, again citing Eden and Kazman (2003): “this lack of distinction is the cause of much muddy thinking, imprecise communication, and wasted, overlapping effort”. One such fundamental and common problem is the problem of architectural erosion. Architectural erosion (Perry and Wolf 1992), also referred to as software erosion and design erosion, is the divergence of the software from the software architecture, caused by violations of the architecture. These violations lead to an increase in problems in the system and contribute to the increasing brittleness of a system that deviates more and more from the desired system qualities, making them increasingly harder to maintain (Bosch 2004, Jansen and Bosch 2005, van Gurp and Bosch 2002).
This problem is addressed by the recent research viewing software architecture as a set of design decisions (described in section 1.2.3). This is a promising but still a relatively immature branch of research, lacking for instance formalisms and tools for enforcement, where success in practice yet remains to be seen.

**ADLs are insufficient for defining architectural design rules.**

The current practice for modelling architectural design rules is to capture them in text and to link them to the resulting design. This may be sufficient for rules stating the existence of elements in the design, such as a subsystem or an interface, since the architect can put the actual element (i.e. a certain subsystem) into the system model at the time of the decision. It is however not sufficient for rules on potentially existing elements such as rules on what kinds of elements, with behavioural and structural rules and constraints, there should be in a certain subsystem, since the actual elements are not known at the time when the design decision is made. Instead, the rule based design occurs later in the detailed design phase, and involves other persons, potentially even in a different version of the system.

ADLs typically allow one to specify components with relations and interfaces together with functional and structural constraints. They do not however provide sufficient means to specify constraints or rules on groups of conceptual components only partly specified by the architect that are intended to be instantiated and detailed by designers. While it is possible to model these kinds of rules in some ADLs such as UML the resulting descriptions are often too complex for practical use. For instance, consider the following simple rule:

\[
A \text{ sensor may only have associations to In\_Port\_Ifc and Data\_Items. These associations shall only be navigable from the sensor.}
\]

In UML this can be modelled by using stereotypes for the different kinds of elements constrained by the rule. The constraints for each kind of element can then be expressed in OCL\(^1\) for each stereotype. The result is shown in Figure 3. There are, however, at

---

\(^1\) OCL, Object Constraint Language, is a language for expressing constraints on UML models standardized by OMG
least two problems with defining architectural design rules in this way. The first problem is that it requires detailed knowledge of the UML meta-model (the model defining the abstract syntax of UML), which is quite complex and beyond what can be expected from a typical architect or developer; thus, it would be likely to impede widespread adoption of the approach (Conboy and Fitzgerald 2010). The second problem is that the OCL expressions become quite complex even for simple rules, as illustrated in Figure 3.

Figure 3. Definition of the architectural design rule example using OCL

Another option could be to use an extensible ADL such as xADL 2.0 (Dashofy 2001). This is an ADL that is defined as a set of XML schemas. The idea is to give the architect the possibility to specialise the constructs of the language or add constructs to the language by writing new XML schemas. Accompanying the XML schemas that define the base language there is a set of tools for extending the language and to use the extended language. xADL can be seen as a variant of DSM, where XML Schemas is used as the meta-modelling language. As is the case for DSM, architectural design rules could be defined by introducing new component types in the language or adding properties to already defined component types. There is however no defined way of constraining already defined component types which means that component types that need to be constrained need to be defined from scratch. Basically this requires the specification of a complete language, a much greater task than just to constrain the use of a general purpose language. Also, currently there is only experimental tool support for xADL, with only limited support for extensions to the base language, which makes the approach unfit for practical use.
As a result of this insufficiency the common practice is to capture architectural design rules in informal text and use error prone and time-consuming manual reviews to enforce them.

2 Scope of Current Work

As described in section 1.1.1, a basic premise of MDD is that all important design information is kept in models and that two important factors for success are that these models are accurate and understandable where accurate means that the models only can be interpreted in one way. As described in section 1.2.4 existing approaches on how to model architectural design rules are either flawed in respect to the accuracy factor or in respect to the understandability factor: While it is possible to model these rules in some ADLs the used formalisms are not easily understandable which has led to the current practice to describe these rules in informal text which breaks the accuracy requirement. This insufficiency is the scope and concern for this thesis and the overarching research objective eventually became:

Establish and evaluate an approach to modelling architectural design rules which can be automatically enforced while also amenable to use by practitioners.

As motivated in subsection 4.2 the defined approach for modelling architectural design rules is based on using UML on the meta-model level to restrict the use of UML in a system model. This means that the approach is limited to support the architectural views that can be modelled with UML. Since UML is a general purpose object-oriented language with the capability of specialising to different domains using stereotypes, this only limits the approach to modelling views where the system is seen as a set of entities communicating using messages or operation calls, where the behaviour is either triggered by operation calls, message receptions or time events. Time-continuous views of the system are however not supported. However, time-continuous architectural views are rare in methods for architectural design, for example all of the architectural views explicitly suggested in the methods listed in subsection 1.2.2 can be modelled in UML:

- The 4+1 view model of architecture (Kruchten 1995) suggests four views: Logical, Process, Physical and Development together with a fifth view, called
Scenarios, showing how the four other views satisfy key use cases. All these views can be modelled in UML.

- The Siemens’ 4 Views (S4V) method (Hofmeister et al. 2000, Soni et al. 1995) suggests four views which all can be modelled in UML: Conceptual structure, module structure, execution structure and code structure.

- Business Architecture Process and Organization (BAPO/CAFCR) (America et al. 2004, van der Linden et al. 2004) suggests five views: Customer, application, functional, conceptual and realization view. All of these can be modelled in UML.

- ADD (Wojcik et al. 2006, Bass et al. 2003), ASC (Ran 2000), QASAR (Bosch and Molin 1999, Bosch 2000, Bengtsson and Bosch 1998) and QADA (Matinlassi et al. 2000) are flexible on which views to use and does not suggest any specific views.

An important part of all architectural design decisions, including those imposing rules and constraints on the detailed design, is the rationale for the decision. In the method for modelling architectural design rules proposed in this thesis there are however no support for modelling the rationale for the rule. The main reason for this is that the rationale is not something that needs to be enforced on the detailed design; the rationale is there to explain the reason for the rule to make sure it is relevant now and in the future. However, the fact that modelling the rationale is out of the scope of the method does not mean that the rationale should not be documented, on the contrary documenting the rationale are just as important using this approach as if not using it. The rationale may be documented in the architectural rules model using UML comments or in a separate document.

As discussed in section 1.2.4 there is no clear separation between architectural design and detailed design in the body of literature. The view adopted in this thesis is that the architectural design defines constraints on the detailed design in order to ensure that the quality attributes of the system are fulfilled. Ultimately it is the task of the architect to decide exactly where to draw the line between the architectural design and the detailed design.
3 Research Method

3.1 Selection of Research Methods

Many research methods exist for use in empirical research in the software field, all of which have their own particular applicability for addressing any given research question. Here the range of research methods chosen for use in this study is presented, their general characteristics, strengths and weaknesses and why they were appropriate for this study are discussed. Further details on the practical operationalization of each of the methods in the particular context are provided in each of the papers in part II of this thesis.

3.1.1 Building theory from case study research

Eisenhardt presents a process for building theory from case study research (Eisenhardt 1989). The process is particularly appropriate when little is known about a phenomenon, current perspectives seem inadequate because they have little empirical substantiation, or they conflict with each other or common sense. Since there are very few studies on work practices regarding architectural rules in the context of MDD and no sufficient solution on the problem on how to model these in the current body of literature the process is a good fit to the research objective for this thesis. The process consists of eight steps, outlined below:

Step 1, Getting Started:

- An initial definition of the research question, in at least broad terms, is important to focus efforts.
- A priori specification of constructs can help to shape the initial design of theory-building research; it provides better grounding of construct measures.
- Begin the research as close as possible to the ideal of no theory under consideration and no hypotheses to test to retain theoretical flexibility

Step 2, Selecting Cases:

- Select a specified population to constrain extraneous variation and sharpen external validity.
- Select cases in which the process of interest is transparently observable to focuses efforts on theoretically useful cases
Step 3, Crafting Instruments and Protocols:
- Multiple data collection methods strengthen grounding of theory by triangulation of evidence.
- Qualitative and quantitative data combined provides a synergetic view of evidence where qualitative data is essential for understanding rationale and for building theory and quantitative data keep researchers from being carried away by false impressions from qualitative data.
- Multiple investigators foster divergent perspectives and strengthens grounding.

Step 4, Entering the Field:
- Overlap in data collection and analysis speeds analyses and reveals helpful adjustments to data collection.
- Flexible and opportunistic data collection methods allow investigators to take advantage of emergent themes and unique case features.

Step 5, Analyzing Data:
- Within-case analysis gains familiarity with data which helps is essential for preliminary theory generation; the overall idea is to become intimately familiar with each case as a stand-alone entity.
- Cross-case pattern search forces investigators to look beyond initial impressions and see evidence thru multiple lenses.

Step 6, Shaping Hypothesis:
- Constantly compare theory and data, iterating toward a theory which closely fits the data.

Step 7, Enfolding Literature:
- Compare the emergent concepts, theory, or hypotheses with the extant literature.

Step 8, Reaching Closure:
- End process when marginal improvement becomes small.
3.1.2 Action case study research

Braa and Vidgen (1999) propose a useful framework (Figure 4) integrating positivist, interpretivist and critical research approaches. Briefly summarising; in Figure 4 the apexes of the triangle represent the different perspectives and outcomes of the research. Thus, from the positivist perspective, a reductionist approach would be followed to produce the desired outcome, which is that of prediction. From the interpretivist perspective, on the other hand, the primary motivation would be that of understanding, while from the critical interventionist perspective, the motivation would be one of change.

![Figure 4. A framework for Integrating Research Perspectives (based on Braa and Vidgen (1999))](image)

As can be seen from Figure 4 the Action Case approach is a hybrid of the Soft Case and Action Research approaches. Soft Case studies is an interpretivist approach to case studies described by Walsham where the concern is with gaining understanding and insight (Walsham 1993). The role of the researcher is that of an observer who does not intervene and avoids affecting the organization. Since both understanding and achieving actual change in the organization were desired outputs from the research presented in this thesis, a pure soft case approach was not deemed appropriate for the research.

Action research originates from the work of Lewin (1948) and several ‘flavours’ have emerged. At heart, however, there is general agreement on a number of essential characteristics: it is a highly participative approach which implies a close intertwining between researchers and practitioners intervening on real problems in real contexts, with two primary outcomes: an action outcome in terms of a (hopefully) beneficial intervention in the organisation, and a research outcome in terms of a contribution to
research on the phenomenon in question. It is also a longitudinal cyclical process of intervention and reflection, with any learning fed back in successive action research cycles (e.g. Baskerville 1999, Kock et al. 1997, Lau 1997). Since the case studies of this research were performed over a limited time-span in different organizations, a full blown action research approach with several full iterations of intervention and reflection was not deemed appropriate.

The Action Case is described by Braa and Vidgen as:

“A trade-off between being an observer who can make interpretations (understanding) and a researcher involved in creating change in practice. With case study methods the researcher aims to collect a rich set of data to provide insight into some situation, while in action research the aim is to support desirable change in an organizational setting. However, when doing case studies researchers contribute to change by questioning events and applying new concepts. On the other hand, full-scale action research projects are often not appropriate due to organizational constraints or the nature of the topic to be investigated. Small scale intervention with a deep contextual understanding is one way of balancing this dilemma—this is the area labelled action case.”

These characteristics were very much present in the two action case studies of this research: They were both a mix of interventionist/change and interpretivist/understanding perspectives. Also, limited intervention was done over a limited time period, and the researcher had deep knowledge of the problem domain and the organizations.

3.1.3 Strategy for searching the literature

Levy and Ellis propose a useful strategy for searching IS literature (Levy and Ellis 2006), the strategy suggests three steps:

- **Keyword search:** This refers to the use of specific words or phrases to search quality scholarly databases to find relevant literature. Keywords can be used against several categories such as keywords, title and abstract. The keyword search should be just the initial, not the main step for a literature search.
• **Backward search:** The process of going backward in the literature can be divided into three sub-steps: *backward reference search, backward authors search* and *previously used keywords.*

  *Backward reference search* refers to the reviewing the references of the articles yielded from the keyword search noted above. This provides researchers with the ability to learn more about the origins of the subject under study. Additional levels of backward search (the references of the references) may also be done to expand knowledge even further.

  *Backward authors search* refers to reviewing what the authors have published prior to the article. Since most IS researchers tend to conduct studies within a rather narrow domain querying an author’s prior work may yield fruitful information.

  *Previously used keywords* refer to reviewing the keywords noted in the articles yielded from previous searches.

• **Forward search:** The third step deals with forward literature search. Similar to the process of backward literature search forward search can be divided into two sub-steps: *forward reference search* and *forward authors search.*

  *Forward reference search* refers to reviewing additional articles that have cited the article. This will enable the researcher to extend their knowledge by locating follow-up studies or newer developments related to the phenomenon under study.

  *Forward authors search* refers to reviewing what the authors have published following the article. Since most IS researchers tend to conduct studies within a rather narrow domain querying an author’s later work may yield fruitful information.

This strategy was used in the following way in the literature reviews presented in this thesis:

• Keyword searches were done in IEEE explore, ACM Portal, ScienceDirect, SpringerLink, Inspec, CiteSeer and Google Scholar.

• Having identified relevant articles, backward and forward reference search and backward and forward author search was performed.

• During the search additional relevant keywords were used to refine the search process.
3.2 Conduction of the Research

The research process, depicted with round-corner boxes and arrows, is shown in Figure 5. The figure also shows in which paper the research is reported. The papers are the contents of the second part of the thesis. The research process is further described in the following text.

Based on experience from more than ten years of work as a software architect in different projects and organizations using MDD, an understanding of problems with how to model and enforce architectural design rules evolved over time. Informed by this experience it was decided to perform a detailed review of the relevant literature. The goals of the review were to investigate the role of architectural design rules, and how these rules can be modelled. An additional goal of the review was to find out what had been reported on architectural practices in large industrial projects using MDD. The literature review revealed a lack of real-world rigorous validation of the MDD approach and an absence of guidelines as to how to model architectural design rules. Given this lack of documented knowledge in the body of literature the overall research objective defined in section 2 was formulated. This corresponds to the first step in Eisenhardt’s process. In line with the process there was an objective defined, an a priori construct to measure; architectural design rules, and there was no hypothesis on solution. The research continued with an action case study (see section 3.1.2) aiming to gain insight...
into work practices regarding architectural design rules in the context of MDD, more specifically to answer the following questions:

- How were the architectural design rules documented?
- How were they communicated and enforced?
- How did the actual work practices regarding architectural design rules affect the overall development process?

In line with Eisenhardt the selected case was an industrial case for developing embedded software where architectural enforcement was an important issue. That is, a case where there were a relatively large number of developers (more than 20) that had to follow an unfamiliar architecture. Multiple data collection methods and both qualitative and quantitative data were used in an opportunistic way where analysis and collection was done iteratively (Eisenhardt step 3 and 4). The analysis was based on a rich set of system and project documentation, collected from the configuration management tool used by the project. Further, the research drew from experience related to participation in the project being investigated. A deep within-case analysis revealed patterns that were compared to experiences from other projects (Eisenhardt step 4).

The findings in the action case study formed a hypothesis on how to model architectural design rules. This was then the basis for a systematic literature review focused on identifying usable ideas to more firmly shape a method. This method was then analysed, refined, formalized and iteratively tested by using it to model the architectural design rules from an existing, previously developed system (Eisenhardt step 6 and 7).

To study the transferability of the approach into practice an action case study was performed where the method supported by a prototypic tool was transferred to a development team in an industrial development project. Of specific interest were the effects on productivity and quality as well as the learnability of the method and tool (Eisenhardt step 6). Semi-structured interviews were used to provide rich information on the interviewees’ prior knowledge, values and expectations, all important for the estimation of the learnability of the method (Lings and Lundell 2004). They were also used to provide information on perceived efficiency and quality of the work done. Two interviews were done with each interviewee. The first, focusing on prior knowledge, values and expectations, was performed before interviewees started to work with the
method. The second interview, focusing on usability, learnability of the method, perceived development efficiency and product quality compared to the traditional way of working, was performed when they had worked with the method for two to six months. Four persons were interviewed - the architect and three developers working with the operating system kernel. Each interview started with a specific set of questions prepared in advance. Many of these were open-ended, intended to solicit information not foreseen by the interviewer and encouraging the interviewee to tell context-rich stories, thus minimising the risk for bias caused by the fact that the interviewer was a cultural insider. The interviews were recorded and then transcribed. The transcribed interviews were then sent to the interviewee who was invited to clarify, correct, and elaborate parts of the interview. All interviewees were also promised anonymity in any reports.

In addition to the interviews archival data was used in the data analysis as a tactic to further minimise the risk of bias from the researcher and improve the validity of the results. The following artefacts produced during development were used as data sources:

- The architectural rules model.
- The system model.
- A text document with the textual rules and additional rationale for the rules

### 3.3 On the Validity of the Results

In this area where little exists by way of documented successful exemplars, then an appropriate approach is an in-depth study which a single case provides, what has been termed the “revelatory case” (Yin 1994). A single case strategy is also strongly recommended by Mintzberg who poses the very apt question: “what, for example, is wrong with samples of one?” (Mintzberg 1979). One of the limitations of the studies for contribution 1 and 3 might appear to be the fact that they are based on single cases and thus there is limited scope for generalisation. However, Lee and Baskerville (2003) identify a fundamental and long-standing problem with the type of generalization based on the type of statistical sampling frequently sought in research, namely the problem of attempting to generalize to any other settings beyond the current one. Following this conventional model, researchers have suggested increasing sample size or number of
case study organizations, but Lee and Baskerville argue cogently for the ultimate futility of this as a “flawed strategy”.

Although statistical generalization is neither feasible nor desirable from a single case study, analytical generalization is possible (Runeson and Höst 2009). Given that this research addresses a gap which has not been addressed in the past in the literature, the results are likely to be of relevance, even if the case study context is different.

Two factors suggest that the results should, to a large extent, be transferable to other systems and organisations in the embedded software domain:

1. The defined transformations are based on raising the general modelling constructs of UML to the meta-model level, not on the specific needs of the system used for the test.
2. The approach is tested on two real-world embedded system of significant size with functionality quite common in this domain.

4 Contribution

The main contributions of this research together with the research methods used to achieve it and the paper (part II of this Thesis) where the related research is presented are listed in Table 1.

<table>
<thead>
<tr>
<th>Contribution</th>
<th>Research Methods used</th>
<th>Paper</th>
</tr>
</thead>
<tbody>
<tr>
<td>Illustrating how insufficient solutions on how to model architectural design</td>
<td>Literature review (3.1.3)</td>
<td>I</td>
</tr>
<tr>
<td>rules are a serious problem in projects using MDD</td>
<td>Action Case study (3.1.2)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Eisenhart theory building process (3.1.1)</td>
<td></td>
</tr>
<tr>
<td>A method, suitable for practice, with prototypic tool support, on how to</td>
<td>Literature review (3.1.3)</td>
<td>II</td>
</tr>
<tr>
<td>model and automatically enforce architectural design rules</td>
<td>Eisenhart theory building process (3.1.1)</td>
<td></td>
</tr>
<tr>
<td>A validation of the method in an industrial development project</td>
<td>Action Case study (3.1.2)</td>
<td>III</td>
</tr>
<tr>
<td></td>
<td>Eisenhart theory building process (3.1.1)</td>
<td></td>
</tr>
</tbody>
</table>
Each contribution is detailed in the following:

1. **An illustration of how insufficient solutions on how to model architectural design rules are a serious problem in projects using MDD.**

   To identify potential improvements to architectural practices in a MDD context an action case study was performed in an industrial case where architectural enforcement was an important issue. The study showed that:
   
   a. Architectural design rules are important design artefacts and there is no sufficient solution in the body of literature on how to model these.
   
   b. The inability to model architectural design rules makes architectural enforcement a bottleneck in MDD projects.

2. **A method, suitable for practice, with prototypic tool support, on how to model and automatically enforce architectural design rules.**

   The method is defined by a set of transformations from UML constructs in the architectural rules model to constraints in stereotypes of a UML profile that is applied to the system model. To ensure that the method is formal enough for automation the transformations is formally defined using the transformation language MOFScript and the constraint language OCL. To further demonstrate that automatic enforcement was viable in a practical setting, tool support adding automatic enforcement to a market leading modelling tool was built. To demonstrate the applicability to real-world problems the method was applied to the architectural rules of an already built industrial strength system. This showed that 88 % of the rules could be modelled and automatically enforced. The rules that could not be modelled were all rules where the developer was supposed to exercise judgement, which made them inherently impossible to formalize.

3. **A validation of the method in an industrial development project.**

   Results from the conduction of an action case study showed high and fast ROI (Return On Investment) of the approach due to low cost for tooling and learning and high productivity gain due to high degree of automation and significant complexity reduction. There was also low resistance from both architect and developers in adopting the method, they all thought that they produced a better result with less effort and had more fun doing it.
Each of these contributions is further discussed in the following three sections.

4.1 Contribution 1: An illustration of how insufficient solutions on how to model architectural design rules are a serious problem in projects using MDD.

To identify potential improvements to architectural practices in a MDD context an action case study was performed in an industrial case where architectural enforcement was an important issue. That is, a case where there were a relatively large number of developers (more than 20) that had to follow an unfamiliar architecture. Questions addressed by this research were:

- How were the architectural design rules documented?
- How were they communicated and enforced?
- How did the actual work practices regarding architectural design rules affect the overall development process?

The project followed a phased process similar to the Rational Unified Process (RUP) according to Figure 6. The main architectural work was done during the Elaboration phase during six months by the two architects. The Construction phase was started with a workshop where the architecture was presented for the design teams. The Construction phase was then performed by approximately 50 developers divided into seven teams during twelve months. To ensure that the design corresponded to the architecture the overall design of each component was reviewed by the architecture team before detailed design of the component was allowed to start. Since there were 166 components, this work occupied the architects almost full time during this phase.
The traditional way of representing the architecture is in a document that guides and constrains the detailed design. In model oriented processes, like RUP, where models have replaced requirement specifications and design descriptions, one still represents the architecture in a document. An aim for the research was to introduce an MDD approach which, insofar as possible, automatically connected the architecture to the design, thereby minimizing both the maintenance problem and the effort to enforce the architecture in the design. In the end the project used the following approach:

- The high level structure was captured in the system model as UML packages.
- Architectural design rules were captured in natural language in a text document supported by a UML class framework in the system model.
- Example components were designed in a package in the system model illustrating how to follow the architectural rules.

Using natural language to describe architectural design rules meant that the project had to rely heavily on manual reviews to enforce conformance with the architecture. Performing these reviews was a heavy burden for the architects; it took almost all of their time during the first 12 months of development after the first release of the architecture.
The rules proved to be ambiguous and hard to comprehend, and thus prone to different interpretations. Several developers reported having a hard time trying to understand and follow all of the detailed rules. This was manifested by the fact that major corrections were frequently needed after reviews, as shown by the review protocols. Sometimes this meant that a lot of reworking had to be done since reviews were often held when design had continued too long. This was caused by work overload on the architects, which in turn was caused by a lot of effort expended on reviewing the designs generated by the different teams. The progress reports show that it was common that architectural reviews were actually done after a component was completed and tested, which was in violation of the rules which stated that the component had to pass the review before the detailed design was done.

4.2 Contribution 2: A method, suitable for practice, with prototypic tool support, on how to model and automatically enforce architectural design rules.

The problem of modelling design rules has much in common with the problem of modelling architectural styles or the solution part of a design pattern in so far as it basically is about specifying rules to follow in the design. An architectural style also known as architectural pattern (Shaw and Garlan 1996) is an idiomatic pattern of system organization. It is comparable to the solution part of a certain kind of design pattern (Gamma 1995) specifying system wide design rules, categorized as architectural patterns in (Buschmann 1996). There are a number of suggestions on how to formally model design pattern specifications and architectural styles (Lauder and Kent 1998, Bayley 2007, Zdun and Avgeriou 2005, Mikkonen 1998, Mak et al. 2004, France et al. 2004, Eden 2002, Pahl et al. 2007). While some approaches use mathematical formalisms such as predicate logic and set theory, others use UML applied at the metamodel level.

An interesting approach in this context is Reflexion Modelling (Murphy 2001). Reflexion Modelling (RM) is a technique to automatically detect differences between design and implementation. In RM the engineer first defines a model that captures the intended design of the system consisting of the main components and their relations. Then the engineer defines a declarative mapping between the design model and the source code artefacts. Finally a tool, such as the SAVE tool (Lindvall 2008), can be used to highlight all deviations in the source code from the intended design in the design

44
model. RM is however not applicable to the problem addressed in this thesis for the following two reasons:

1. In MDD the source code is generated from the design model so the problem of source code deviating from design does not exist.
2. Even if we could use the RM technique to compare an architectural model with a design model this would not help. RM is only applicable to the part of the architecture that defines the existence of specific components and their relations, e.g. that a certain subsystem should exist and have a specific interface. This part of the architecture is not a problem in MDD since it can be directly represented in the system model. The problem addressed by the method of this thesis is the problem of how to represent and enforce rules on conceptual components where the actual components are to be identified and defined in the detailed design, e.g. rules that apply to a certain kind of classes, such as all classes representing sensors, in a UML model (see also section 2.3 in Paper I).

Other interesting approaches in this context are DSM and the software factory approach (see subsections 1.1.3 and 1.1.4). They are interesting in that they allow you to naturally specify rules on the system model in the DSML definition. In fact, that is basically what the DSML definition is - a set of rules that have to be followed when building the system model. A problem with these approaches is however that they require the specification of a complete language, a much greater task than just to constrain the use of a general purpose language. However, in a situation where an organisation has decided to use a DSM approach, the architectural design rules could probably be a subset of the rules of the DSML definition.

In order to be successful in practice, it is essential that architectural design rules are modelled in such a way that they are both amenable to automatic enforcement of the detailed design and easy to understand and use by both architects and developers (section 1.1.1: models must be accurate, understandable and inexpensive (Selic 2003)). The latter is important in order to avoid increasing the work of developing the rules; otherwise there is a risk that the work burden is increased instead of decreased even though the enforcement is automated. Another important issue is that it should be possible to use current mainstream modelling tools to model both the architectural
design rules and the system model so as to make it widely adoptable. Given its widespread diffusion, UML would be a good choice for modelling architectural design rules. Such an approach, illustrated in Figure 7 is presented in this thesis.

![Figure 7](image)

**Figure 7.** An approach for automatic enforcement of architectural design rules.

The approach is based on the same idea as in Zdun and Avgeriou (2005) and France et al. (2004), namely to use UML on the meta-model level to restrict the use of UML in a system model. However, instead of using it to specify patterns, it is used to specify architectural design rules. Classes in the architectural rules model are transformed into UML stereotypes carrying constraints given by the constructs of the architectural rules model. Using this approach the constraints defined in Figure 3 in section 1.2.4 are modelled according to Figure 8, as can be seen this is significantly less complex and does not require any knowledge on the UML meta-model.

![Figure 8](image)

**Figure 8.** The rules in Figure 3 modelled according to the approach presented in this thesis.

Another alternative could have been to use a meta-modelling language with explicit elements for relations and roles such as the GOPRR (Kelly 2008) meta-modelling language. This would however not have made modelling simpler (as illustrated in Figure 9) and would also have violated the need for a widespread language to ease adoption.
To demonstrate the applicability of the method to a real problem it was applied to an already developed system with an existing set of textual architectural rules, a

---

2 http://www.eclipse.org/gmt/mofscript/
representative subset of these rules are shown in Table 2. The rules could be classified into three categories: structural, behavioural and judgmental. Structural rules specified structural constraints such as rule 4.1, 4.2, 6.7 and 8.1 in the table. Behavioural rules specified constraints on behaviour such as rule 9.16 in the table. Judgemental rules were rules where the developer had to exercise judgement to follow the rule; 3.2 is an example of such a rule in the table. There were 66 rules in total; eight of these could not be modeled. These were all judgmental and therefore inherently impossible to formalize.

Table 2. A subset of the original architectural design rules

<table>
<thead>
<tr>
<th>Id</th>
<th>Original rule (Quotation)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2</td>
<td>“Functionality specific to a PAPI requirement shall be kept in this layer unless it is reusable for another PAPI or applicable to DVB standard. In this case it shall be placed in CMP or CMD.”</td>
</tr>
<tr>
<td>4.1</td>
<td>“All coupling between arcComponents shall be loose in the sense not statically linked”</td>
</tr>
<tr>
<td>4.2</td>
<td>“All associations between arcComponents shall be navigable from the client to the server (user to the resource)”</td>
</tr>
<tr>
<td>6.7</td>
<td>“In the case of a component locked to a specific arcComponentUser, it is the responsibility of the locker to allow only one thread at a time to access the component.”</td>
</tr>
<tr>
<td>8.1</td>
<td>“All locked components shall inherit the same base class, arcLocked.”</td>
</tr>
<tr>
<td>9.16</td>
<td>“Transmission events and exceptions initiated by a Write() shall be reported back to the arcPortUser via the TxDone() call.”</td>
</tr>
</tbody>
</table>

4.3 Contribution 3: A validation of the method in an industrial development project.

To build an understanding of the effects of using the method in an industrial development project an action case study was performed. Of specific interest were the effects on productivity and quality as well as the learnability of the method and tool.

The action case study was performed in a development project at Saab Avitronics, a business unit within the Swedish defence industry Saab (www.saabgroup.com). The purpose of the development project was to develop a software platform, including a real-time operating system with special scheduling features, to be used in some of the system computers of a new version of the Gripen fighter aircraft. The project started in the spring 2009 and had its first delivery at the summer of 2010. It involved one project
manager, one software architect and five developers. The study started in March 2009 and ended in February 2010.

The development project started with an initial phase during March 2009 where the initial architectural rules were modelled in collaboration between the method developer and the architect. Then this architecture model was evolved by the architect alone until the beginning of August 2009. At that point a four-hour workshop was held where the rules were presented for the developers by the architect. The developers then started the detailed design according to the architecture. Three incremental iterations were done in February 2010 and a first delivery of the system was planned for mid-2010.

4.3.1 Effects on the development process

The changes made to the development process of the company due to the introduction of the new approach for architectural modelling and enforcement is illustrated in Figure 10.
The traditional process of the company starts with the definition of the architecturally significant requirements; these are essentially the quality requirements. Then the architect undertakes the architectural design addressing the architecture requirements. Essentially the architect makes a number of decisions on how the system should be designed in order to ensure that architectural requirements are fulfilled (see section 1.2.2). The result of this is typically a high-level structure of the system, a framework of base-classes implementing the communication infrastructure of the system, and a set of structural and behavioural rules to follow in the design of the components that will populate the system model. The high level structure and the framework go into the system-model and the rules are specified in a textual document. During the architectural design the architecture is regularly validated against the architecture requirements and if necessary the requirements are renegotiated. When the architecture has been validated against its requirements the next development phase begins. In this phase the system is
developed in a number of increments with increasing functionality. Each development increment begins with a definition of the requirements to be implemented in the increment. Then each development team does a preliminary design defining how the required functionality is allocated to new and existing components according to the architectural design rules. Before the development team is allowed to start the detailed design of the components, the preliminary design must pass the architectural review. In the architectural review the architect checks that the design does not violate the architectural design rules. During preliminary design and detailed design the architect supports the development teams and guides them in how to follow the architectural design rules. If any problems with the architecture revealed during preliminary design necessitate changes to the architecture, these changes are done by the architect in parallel with the development work.

In the new process used in the case study project the most notable difference to the traditional process was that the manual architectural review was removed. Since all but five (out of 126) architectural rules were automatically enforced it was enough for the architect to double-check that the remaining five rules had been followed at the end of the iteration. This removed the architect as a bottleneck in the development and increased significantly the amount of time available for guiding the teams. Another difference was that the architect modeled the architectural design rules instead of expressing them as informal text which also increased the quality of the architecture and made it simpler to communicate.

4.3.2 Effects on productivity

The architect states that even though he was forced to develop both textual rules and the model, due to process requirements, he gained efficiency in developing the architecture. This was due to the fact that the model was at a higher level of abstraction than the textual rules, making the complexity lower, and that the formalism of the models eliminated ambiguous and contradicting rules. It was easier, more efficient and gave better results to first model the rules and then manually extract the textual rules from the model, than the traditional practice to work with rules only as text. He estimates that in a normal situation where the textual rules would not be needed he would gain between 10 to 50% depending on the complexity of the rules, in this phase. The greatest effect for his productivity though, he estimated, was that he had to spend a lot less time
communicating and reviewing the design than in a normal MDD project. Normally he would spend one to two days doing an architectural review (see Figure 10). This time was now reduced to less than an hour checking the five rules not checked by the tool. He also used to spend a large part of his time explaining the meaning of the rules to the teams, with this new way of working there was almost no such questions. Spending a lot less time on reviews and communicating the rules enabled him to do optimisations to the architecture and supporting the teams in their design which, he stated, had made the system better and the teams more efficient.

All developers claimed that they found it faster to read and understand the modelled rules than the textual rules. The only difficulty they saw was that the error messages from the tool sometimes were difficult to understand, but as one of them pointed out:

“This is the same problem as you have with a new compiler; the error messages can be hard to understand at the beginning but you soon learn how to interpret them.”

All developers believed that after a few days they were already more efficient then they would have been if working the traditional way, and were sure that the quality, in respect to how they had followed the rules, was much higher than if they would have had to manually follow the textual rules. This, they estimated, had an even greater impact on their productivity since there was no longer any rework after the architectural reviews.

4.3.3 Effects on quality

According to the architect, when working in the traditional way where the architectural rules are enforced by manual reviews, generally a few violations are not detected until very late in the process, typically in the design of a future version of the system, since many architectural rules address maintainability, scalability and portability requirements. With automatic detection of all violations there are no violations to the architectural design rules in the system. According to Boehm and Basili (2001):

“Finding and fixing a software problem after delivery is often 100 times more expensive than finding and fixing it during the requirements and design phase.”

This means that eliminating architectural violations should have a major impact on the quality of the system. The architect also claimed that the rules were of higher quality,
compared to textual rules, since there were no ambiguous, redundant or contradicting rules due to the formal modelling.

Another effect was that both the review and the discussions about the architecture between the architect and the developers focused on how to package the functionality in different components and not on how to follow detailed design rules since this latter question was automatically handled. The architect claimed that in a “normal” project this would have been the opposite way around and that he believed that the increased focus on more difficult design decisions also led to a better system.

The only apparent risk was that of relying too much on the tool and missing things that were not checked by the tool either by design or because of possible errors in the tool.

4.3.4 Learnability of the method

The architect had eleven years of experience from UML modelling working as an architect in several organizations within the embedded software industry. He had also quite deep knowledge of the UML metamodel from building model-to-code transformations in MDAWorkbench. The architect was positively disposed to using the approach since he was the one who had taken the initiative to use it after having seen a presentation of the method and tool at an internal workshop arranged for technology leaders within Saab. Developer A had a masters degree in mechatronics and automation and two years of experience of design and implementation of embedded software. Developer B had a bachelor degree in electronics and three years of experience in design and implementation of embedded software. Developer C had a master degree in software engineering and electronics and two-and-a-half years of experience of design and implementation of embedded software. None of the developers had any knowledge of metamodelling and all had only a one-week training course in UML modelling. All developers were positively disposed to using the method and tool based on expectations that it would reduce the effort needed to follow the architectural rules.

The architect and the developer of the method collaborated in developing an initial version of the architectural rules model with an effort of about two person-weeks during March 2009. After that the architectural rules model was evolved by the architect alone until the beginning of August with an effort of about three person-weeks. At that point
the model was presented to the developers in a four-hour long workshop. After that there have only been small adjustments to the model. The architect estimates that it took about one month to master most of the modelling concepts and an additional month to fully master the method.

Both the developers and the architect emphasized the benefits of getting immediate feedback from the tool, it made the learning of the rules much faster. All three developers claimed that they found it easy to understand the modelled rules right from the beginning with the primary reasons being that the rules were modelled in UML, which they all had some experience of, and that they got instant feedback from the validation tool, illustrated by these statements:

- “If you understand UML then you understand the rules”
- “Instant feedback is the only efficient way to learn. If you give a banana to a monkey one hour after it has done something good nothing happens, if you do it instantly, it will learn to do it again”

5 Summary and Discussion

The first contribution of this research is an illustration of how insufficient solutions on how to model architectural design rules are a serious problem in projects using MDD. The contribution was made by conducting a case study of an industrial case where architectural enforcement was an important issue. The action case study showed that architectural design rules are an important part of the architecture and that before this research there were no satisfactory solutions on how to model them in the existing body of literature. The inability to formalize the architectural design rules leads to a need for manual enforcement of them. The action case study showed that this is an error-prone and time-consuming task that may require most of the effort of the architects during the construction-intensive phases of a project. This problem exists in traditional document-based development as well as in MDD but it is more apparent and acute in MDD. This is because MDD has been able to automate the step from detailed design to implementation eliminating time consuming coding and code reviews. However, MDD has not been able to automate enforcement of the architecture on the detailed design due to the inability to model architectural design rules. The action case study showed that the inability to model architectural design rules makes architectural enforcement a bottleneck in MDD projects. This leads to a plethora of problems, including:
- **Stalled detailed design:** The design teams have to wait for the architects to review their overall design before they can dig deeper into the design.

- **Premature detailed design:** Design teams start detailing their design before their overall design is approved by the architect, with the risk that they will have to redo much work after the review. In the case study the progress reports show that it was common that architectural reviews were actually done after a component was completed and tested.

- **Low review quality:** Low quality of the reviews, leading to problems later in the project. In the case study the architects frequently complained about having too little time to perform the reviews with good quality. Review protocols show that in many cases violations to the architecture actually was made in an earlier version of the component, but had been missed in the previous review; this typically resulted in the need of major corrections to the component.

- **Poor communication of the architecture:** The architects have no time to handle the communication with the design teams regarding architectural interpretations or problems, problems are “swept under the carpet.” In the case study several developers reported having a hard time trying to understand and follow all of the detailed rules. This was manifested by the fact that major corrections were frequently needed after reviews, as shown by the review protocols.

To address this problem a method for modelling architectural design rules together with a prototypic tool for enforcing them was developed. This is the second contribution of the thesis. The method was designed to be **Inexpensive, Accurate** and **Understandable**, all factors of paramount importance for successful MDD (Selic 2003) as discussed in section 1.1.1:

- **Accurate** means that the model is formal enough to allow automation.

- **Understandable:** An important property of the approach is that the architectural design rules are modelled using UML at a high abstraction level, without requiring detailed knowledge of the UML meta-model. That the rules are modelled at an abstraction level close to that of the rule itself is required for the models to be easily understandable for architects and developers.
Introduction

- **Inexpensive:** The use of UML reduces the required investment in tools and training since architects and developers benefit from previous knowledge in UML and are able to use their current UML tools for modelling. This dimension of inexpensive is primarily important for acceptance from management. Another dimension of inexpensive is that the method reduces the effort for enforcement, development, maintenance and for following the rules. This dimension of inexpensive is important for acceptance from architects and developers.

The method was based on the idea of using UML to model concepts with behavioural and structural constraints at the meta-model level and then transforming these to a UML-Profile to constrain the system model. The core of the method was a set of formally defined transformations from UML constructs used in the architectural rules model to stereotypes with constraints in a profile to be applied on the system model. The transformations was defined both formally using the transformation language MOFScript\(^3\) and informally by text and examples. The method was developed in an iterative process where it was gradually formed and tested against the architectural rules of an already developed system.

The third contribution of this thesis is a validation of the method in an action case study of an industrial development project. The study was performed in a project to develop a software platform to be used in a fighter aircraft. The study started with an initial phase during March 2009 where the initial architectural rules were modelled in collaboration between the researcher and the software architect of the project. Then this architecture model was evolved by the architect alone until the beginning of August 2009. At that point a four-hour workshop was held where the rules were presented for the developers by the architect. The developers then started the detailed design according to the architecture. Three incremental iterations were done in February 2010 and a first delivery of the system was planned for mid-2010.

Interviews were used as the primary data source of the study. Four persons were interviewed - the architect and three developers working with the operating system kernel. Two interviews were done with each interviewee. The first, focusing on prior

\(^3\) http://www.eclipse.org/gmt/mofscript/
knowledge, values and expectations, was performed before interviewees started to work with the method. The second interview, focusing on usability, learnability of the method, perceived development efficiency and product quality compared to traditional way of working, was performed when they had worked with the method for two to six months. Archival data in the form of models and other development artefacts, as detailed in section 3.2, was used as a second data source. In order to increase the validity of the results the two data sources were used for triangulation.

The study shows that the approach significantly improves productivity and quality in an industrial development project. One important reason for this was that a high percentage of the rules could be modelled and automatically enforced; in fact, all rules not requiring the developers to exercise their judgment could be modelled. Another important reason was that the models eased understanding of the rules; it required less effort both to construct and use them. Also we found that the approach was easy to learn, especially for the developers. The demands are clearly much higher for the architect, who is tasked with constructing the rules, so it is not surprising it takes longer to master the method, still, two months must be considered to be a relatively short time to fully master a new modelling technique. It probably takes longer without any knowledge of metamodelling but judging from how fast the developers, without any metamodelling experience, understood the models this should not be a major hurdle. A lack of experience in UML modelling (or more significantly, any OO modelling language) would probably be a bigger problem, but UML modelling are today to be considered to be basic required skills from a software architect. An important factor for rapid learning of the approach, reported by all interviewees, is the fast feedback on violations which the tool provides.

An important aspect for any technology to be used in practice is its ROI (Return On Investment), the ratio between the cost and benefit in using the technology. The cost in this case is low since very little training is needed and very small investment in tools is required. This makes the ROI high and fast even with a small gain. The benefit, however, can be expected to be considerable since architectural reviews can be a bottleneck in the MDD process, especially for large systems, so the ROI introducing this approach can be expected to be high and the payback time fast.
However, since change in work practice requires people to change, perhaps the most important benefit of the method was the enthusiasm expressed both by the architect and the developers using it, they all thought that they produced a better result with less effort and had more fun doing it. As expressed by one of the interviewees

“I think you have come up with something really useful here.”

Although there is nothing in the approach that explicitly limits it to the embedded software domain, it has only been applied in this domain. There is therefore a need for further research to study the implications when adopting the approach in other application domains. Factors to investigate include the ease with which architects, developers and other stakeholders can learn the approach and accommodate their working practices to it. There is also a need for further research on architectural rules modelling in MDD outside the context of using UML for system modelling, such as when Domain Specific Modelling is used.

6 Organization of the rest of this Thesis

The rest of this thesis consists of three papers as illustrated in Figure 5: Paper I presents a systematic review of the literature to investigate the role of architectural design rules and investigate the application of MDD in a large project to assess the extent to which architectural design rules could be smoothly integrated into the process. Paper II defines an approach on how to model architectural design rules together with a tool for automatic validation of a design model against the architectural design rules. Paper III reports on validation of the method in an industrial development project.

References


61


Liskov, B. and Zilles, S. (1974) 'Programming with abstract data types', in *Proceedings of the ACM SIGPLAN symposium on Very high level languages*, Santa Monica, California, United States, ACM,


OMG (2003a) *MDA Guide version 1.0.1*, OMG.


Zdun, U. and Avgeriou, P. (2005) 'Modeling architectural patterns using architectural primitives', in *20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, San Diego, CA, USA, ACM,
Part II

The Papers
Linking Model Driven Development and Software Architecture: A Case Study

Anders Mattsson\textsuperscript{1}, Björn Lundell\textsuperscript{2}, Brian Lings\textsuperscript{2}, Brian Fitzgerald\textsuperscript{3}

1. Combitech AB, Jönköping, Sweden
2. University of Skövde, Skövde, Sweden
3. University of Limerick, Limerick, Ireland


Research and writing: Anders Mattsson
Critical revision: Björn Lundell, Brian Lings, Brian Fitzgerald
Abstract

A basic premise of Model Driven Development (MDD) is to capture all important design information in a set of formal or semi-formal models which are then automatically kept consistent by tools. The concept however is still relatively immature and there is little by way of empirically validated guidelines. In this paper we report on the use of MDD on a significant real-world project over several years. Our research found the MDD approach to be deficient in terms of modelling architectural design rules. Furthermore, the current body of literature does not offer a satisfactory solution as to how architectural design rules should be modelled. As a result developers have to rely on time-consuming and error-prone manual practices to keep a system consistent with its architecture. To realise the full benefits of MDD it is important to find ways of formalizing architectural design rules which then allow automatic enforcement of the architecture on the system model. Without this, architectural enforcement will remain a bottleneck in large MDD projects.
1 Introduction

Model-Driven Development (MDD) [45] is an emerging discipline [4], [44] where the prevalent software-development practices in industry are still immature [17]. The success of MDD in practice is currently an open question [17] and there is a lack of proven real-world usage of MDD in large industrial projects.

A basic premise of MDD is to capture all important design information in a set of formal or semi formal models that are automatically kept consistent by tools. This raises the level of abstraction at which the developers work which can eliminate time-consuming and error-prone manual work in keeping different design artefacts consistent [17]. An important design artefact in any software development project, with the possible exception of very small projects, is the software architecture [6]. An important part of any architecture is the set of architectural design rules. We define architectural design rules as the rules, specified by the architect(s), which need to be followed in the detailed design of the system. The importance of architectural design rules has been further highlighted by recent research on software architecture [20], [21], [24], [26], [50] which has acknowledged that a primary role of the architecture is to capture the architectural design decisions. An important part of these design decisions consists of architectural design rules. In a MDD context the design of the system is captured in models of the system, therefore architectural design rules in a MDD context specify rules that the models of the system have to conform to. There is however no satisfactory solution in the current body of literature on how to model architectural design rules.

Given the above concerns – the lack of real-world rigorous validation of the MDD approach, and the absence of guidelines as to how architectural design rules can be modelled – our research objective here was to investigate the application of MDD in a large project and to assess the extent to which architectural design rules could be smoothly integrated into the process.

This paper is organized as follows. In the next section we review the literature on architectural design rules, and especially research relevant to modelling architectural design rules and MDD. In section three we present the research approach adopted in this study. In section four we present the findings of a case study and finally in section five we discuss our conclusions and implications for theory and practice.
2 Modelling Architectural Design Rules and MDD

In order to validate our research objective, we conducted a detailed review of the relevant literature, using the specific approach proposed in [32].

- Keyword searches were done in IEEE explore, ACM Portal, ScienceDirect, SpringerLink, Inspec, CiteSeer and Google Scholar.
- Having identified relevant articles, we performed backward and forward reference search and backward and forward author search.
- During the search additional relevant keywords were used to refine the search process.

The primary goal of the review was to investigate the role of architectural design rules, and how these rules can be modelled, especially in relation to MDD. We also wanted to find out what had been reported on architectural practices in large industrial projects and how this related to our case study. Overall, there are very few case studies on model driven development in industrial projects e.g. [3], [48] but to the best of our knowledge there are no case studies that illuminate architectural work practices in such projects. The main conclusions of this literature review were:

- Architectural design rules are important design artefacts for which there is no direct support in MDD.
- There is no satisfactory solution on how to model architectural design rules in the current body of literature.

We discuss these findings in more detail in the sub-sections below.

2.1 The role of Architectural Design Rules

IEEE has established a set of recommended practices for the architectural description of software-intensive systems [19] which are followed by several architectural design methods:

- Attribute-Driven Design (ADD) [6], [53] developed at the CMU/SEI;
- The Rational Unified Process® (RUP) 4 + 1 views [25], [27] developed and commercialized by Rational Software, now IBM;
- The QASAR method [8], [9], [10] developed by the RISE research group at the University of Karlskrona/Ronneby;
- Siemens’ 4 Views (S4V) method [18], [47], developed at Siemens Corporate Research;
- Business Architecture Process and Organization (BAPO/CAFCR) developed primarily at Philips Research [1], [52];
- Architectural Separation of Concerns (ASC) [43] developed at Nokia Research;

The purpose of the architecture is to guide and control the design of the system so that it meets its quality requirements. Bass et al. [6] are unequivocal in stating the importance of an architectural approach:

“The architecture serves as the blueprint for both the system and the project developing it. It defines the work assignments that must be carried out by design and implementation teams and it is the primary carrier of system qualities such as performance, modifiability, and security – none of which can be achieved without a unifying architectural vision. Architecture is an artefact for early analysis to make sure that the design approach will yield an acceptable system. Moreover, architecture holds the key to post-deployment system understanding, maintenance, and mining efforts. In short, architecture is the conceptual glue that holds every phase of the project together for all of its many stakeholders.”

A common understanding in architectural methods is that the architecture is represented as a set of components related to each other [42], [46]. The components can be organized into different views focusing on different aspects of the system. Different methods propose different views; typical views are a view showing the development structure (e.g. packages and classes), a view showing the runtime structure (processes and objects) and a view showing the resource usage (processors and devices). In any view each component is specified with the following:

- an interface that documents how the component interacts with its environment.
- constraints and rules that have to be fulfilled in the design of the component.
- allocated functionality.
- allocated requirements on quality attributes.
A typical method of decomposition (see for instance [6], [53] and [9]) is to select and combine a number of patterns that address the quality requirements of the system and use them to divide the functionality in the system into a number of elements. Child elements are recursively decomposed in the same way down to a level where no more decomposition is needed, as judged by the architect. The elements are then handed over to the designers who detail them to a level where they can be implemented. For common architectural patterns such as Model-View-Controller, Blackboard or Layers [13] this typically means that you decompose your system into subsystems containing different kinds of classes (such as models, views and controllers). However the instantiation into actual classes is often left to the detailed design, for two main reasons:

1. Functionality will be added later, either because it was missed or because a new version of the system is developed, so more elements will be added later that also have to follow the design patterns decided by the architect.
2. It is not of architectural concern. The concern of the architect is that the design follows the selected architectural patterns, not to do the detailed design.

This means that a substantial part of the architecture consists of design rules on what kinds of elements, with behavioural and structural rules and constraints, there should be in a certain subsystem.

The importance of architectural design rules is also highlighted in current research in software architecture which is focused on the treatment of architectural design decisions as first class entities [20], [21], [24], [26], [50], where architectural design decisions impose rules and constraints on the design together with rationale. However, there is not yet any suggestion on how to formally model these design rules. The current suggestion is to capture them in text and to link them to the resulting design. This may be sufficient for rules stating the existence of elements (“ontocrisis” in [24]) in the design, such as a subsystem or an interface, since the architect can put the actual element (i.e. a certain subsystem) into the system model at the time of the decision. It is however not sufficient for rules on potentially existing elements (“diacrisis” in [24]) such as rules on what kinds of elements, with behavioural and structural rules and constraints, there should be in a certain subsystem, since the actual elements are not known at the time when the design decision is made. Instead, the rule-based design occurs later in the
detailed design phase, and involves other persons, potentially even in a different version of the system.

### 2.2 MDD and Architectural Design Rules

The basic idea of MDD is to capture all important design information in a set of formal or semi formal models that are automatically kept consistent by tools. The purpose is to raise the level of abstraction at which the developers work and to eliminate time consuming and error prone manual work in keeping different design artefacts consistent [17].

MDD requires that the work products produced and used during development is captured in models to allow automation of non-creative tasks such as transformation of models into code or conformance checks between different design artefacts. There exist several approaches to Model-Driven Development (MDD) such as OMG’s MDA [40], Domain Specific Modelling (DSM) [22], [49], and Software Factories [16] from Microsoft.

MDA prescribes that three models, or sets of models, shall be developed as illustrated in Fig. 1:

![Fig. 1. An overview of MDA](image)

1. The Computational Independent Model(s) (CIM) captures the requirements of the system.
2. The Platform Independent Model(s) (PIM) captures the systems functionality without considering any particular execution platform.

3. The Platform Specific Model(s) (PSM) combines the specifications in the PIM with the details that specify how the system uses a particular type of platform. The PSM is a transformation of the PIM using a mapping either on the type level or at the instance level. A type level mapping maps types of the PIM language to types of the PSM language. An instance level mapping uses marks that represent concepts in the PSM (such as a process or a CORBA object). When a PIM shall be deployed on a certain platform the marks are applied to the elements of the PIM before the transformation.

MDA does not prescribe any particular language to be used for the models, but UML [41] is proposed as one possibility. There is also an accompanying OMG standard, MOF (Meta-Object Facility), which can be used to describe meta-models for modelling languages. MDA does not directly address architectural design or how to represent the architecture but the architecture has to be captured in the PIM or in the mapping since the CIM captures the requirements and the PSM is generated from the PIM using the mapping.

Another approach to MDD is Domain Specific Modelling (DSM). The basic idea of Domain Specific Modelling (DSM) is that instead of using a general purpose language such as UML to model your system you define and use a language that is specifically well suited to define systems in a narrow domain. In Fig. 2 the basic approach of DSM is illustrated. A Domain Specific Modelling Language (DSML) is defined that captures the main concepts in the domain of an application. This DSML definition is then used as input into a language configurable modelling tool in which the system is modelled in the DSML. Examples of this approach are described in [22] and [49].

The DSML is defined by an abstract syntax, a concrete syntax, semantics, a mapping between the abstract syntax and the concrete syntax and a mapping between the abstract syntax and the semantics. The abstract syntax is typically defined in a model that defines the concepts of the domain, relationships and integrity constraints. For this one can, for instance, use UML and OCL. Since this is a model which in turn defines elements of a language for a model it is called a meta-model [2]. The concrete syntax of
the language comprises the visual symbols that represent the concepts in the metamodel. To provide this and the mapping to the abstract syntax one must specify symbols for the model elements in the metamodel. The semantics of the language define the meaning of the concepts in the abstract syntax. Defining semantics and mapping these to the abstract syntax can be done by providing translations of the metamodel into another language with defined semantics, such as a programming language. DSM can be seen as a special case of MDA where a domain specific language is defined for use in the PIM. Software factories is a MDD approach that incorporates the DSM idea but goes further since it provides a method for building complete customized tool chains for product families using extensible tools. Such a tool chain is called a software factory.

Although neither the DSM nor the software factory approach directly address the problem of how to model architectural rules it is interesting in that they allow you to naturally specify rules on the system model in the DSML definition. In fact, that is basically what the DSML definition is - a set of rules that have to be followed when building the system model. These rules are however not the architectural design rules, they are the rules of a domain specific language.

![Diagram](image)

Fig. 2. Domain Specific Modelling

### 2.3 Modelling Architectural Design Rules

There are a large number of Architectural Description Languages (ADL) [34], [35], [36], including UML, specified for describing the architecture of software systems. These typically allow one to specify components with relations and interfaces together
with functional and structural constraints. They do not however provide any means to specify constraints or rules on groups of conceptual components only partly specified by the architect that are intended to be instantiated and detailed by designers. For instance, in the project reported on in this study, the architects needed to specify a set of rules on behaviour and relations on a conceptual component called arcComponent without knowing which specific arcComponents would be relevant. Rather, they were to be identified and designed by the designers according to the rules stated by the architects.

The problem of modelling design rules is essentially the same problem as modelling the solution part of a design pattern since the solution specifies rules to follow in the design. There are a number of suggestions on how to formally model design pattern specifications [7], [14], [29], [33], [37]. They are however all limited in what kind of rules they can formalize, typically only structural rules. In addition all approaches except [33] require the architect to use mathematical formalisms such as predicate logic and set theory that may be unfamiliar or hard to understand both for architects and developers.

There are also some approaches on how to model the usage of architectural design patterns or architectural styles in a system model such as [38], [55]. However, they only address the problem of how to show that an architectural rule has been followed, not the problem of how to specify the rule.

3 Research approach

Much of the research on the application of software methods to date has relied on postal surveys to investigate factors using statistical techniques. This is undoubtedly useful, but it is often beneficial to complement this with ‘thick’ description which provides a more detailed and nuanced description of the factors at play in a particular context.

This research is based on experience at Combitech AB (www.combitech.se), a Swedish supplier of services within system development, system integration, information security and system safety. Combitech, a wholly-owned subsidiary of Saab AB, can be found in 20 locations in Sweden and the company has more than 800 employees. Customers are primarily from the defence and telecommunications industry, as well as government offices and authorities responsible for infrastructure in society.

Braa and Vidgen [11] propose a useful framework (Fig. 3) integrating positivist, interpretivist and critical research approaches. Briefly summarising, in Fig. 3 the apexes
of the triangle represent the different perspectives and outcomes of the research. Thus, from the positivist perspective, a reductionist approach would be followed to produce the desired outcome, which is that of prediction. From the interpretivist perspective, on the other hand, the primary motivation would be that of understanding, while from the critical interventionist perspective, the motivation would be one of change.

Given the lack of research to date on the application of MDD in real industrial projects, and more specifically the modelling of architectural design rules, this study was concerned with achieving an increased understanding of this process. Bearing this in mind, an interpretivist approach which sought to inductively develop a richer understanding based on a deep analysis of a single case was deemed appropriate. Also, as it represents uncharted territory to a large extent, the study was also motivated by an interventionist desire to achieve successful change in this real organizational problem given the lack of any roadmap documenting how this can be successfully achieved.

Given these objectives, a hybrid of the interventionist/change and interpretivist/understanding perspectives was appropriate. Braa and Vidgen locate a number of hybrid research approaches where a mixture of perspectives is motivating the research, and in case of a mixture of interventionist/change and interpretivist/understanding perspectives, as in this study, the Action Case approach is deemed appropriate. As can be seen from Fig. 3, the Action Case approach is a hybrid of the Soft Case and Action Research approaches, each of which are discussed in turn below.

In Soft Case research an interpretivist approach is adopted. The concern is more with gaining understanding and insight [51]. It is our belief that in this area where little exists by way of successful exemplars, then the appropriate approach is an in-depth study.
which a single case provides, what has been termed the “revelatory case” [54]. A single case strategy is also strongly recommended by Mintzberg who poses the very apt question: “what, for example, is wrong with samples of one?” [39]. One of the limitations of this study might appear to be the fact that it is based on a single case and thus there is limited scope for generalisation. However, Lee and Baskerville [30] identify a fundamental and long-standing problem with the type of generalization based on the type of statistical sampling frequently sought in research, namely the problem of attempting to generalize to any other settings beyond the current one. Following this conventional model, researchers have suggested increasing sample size or number of case study organizations, but Baskerville and Lee argue cogently for the ultimate futility of this flawed strategy.

Action research originates from the work of Lewin [31] and several ‘flavours’ have emerged. At heart, however, there is general agreement on a number of essential characteristics: it is a highly participative approach which implies a close intertwining between researchers and practitioners intervening on real problems in real contexts, with two primary outcomes: an action outcome in terms of a (hopefully) beneficial intervention in the organisation, and a research outcome in terms of a contribution to research on the phenomenon in question. It is also a longitudinal cyclical process of intervention and reflection, with any learning fed back in successive action research cycles e.g. [5], [23], [28].

These characteristics were very much present in this research: The primary author is currently Lead Engineer in Model-Driven Development and Software Architecture at Combitech. He has 18 years of experience from development of embedded real-time systems from a wide range of organizations in the automotive, defence, medical, telecom and automation industries. The last 13 of these years he has alternated between the roles of a software architect and a mentor in software architecture and Model-Driven Development in several projects. The research reported in this paper was motivated by problems in architectural work practices experienced in these projects.

The overall objective of this research was to identify potential improvements to architectural practices in a MDD context based on tensions between MDD practices and architectural practices as revealed in an industrial case where architectural enforcement was an important issue. That is, a case where there were a relatively large number of developers (more than 20) that had to follow an unfamiliar architecture. Questions to be answered by this research were:
How were the rules documented?
How were they communicated and enforced?
How did the actual work practices regarding architectural design rules affect the overall development process?

The research focused on architectural practices in a development project that was executed over a two-year period. The analysis is based on a rich set of system and project documentation, collected from the configuration management tool used by the project, including:

- The architecture design rules document and the system model revealing how the architecture was documented and how the detailed design was done.
- Documents defining the development process of the project.
- Architecture review protocols with comments and actions revealing problems in the interpretation of the architecture and the effort to correct them. There were 120 review protocols with an average of 13 remarks in each protocol.
- Project plans and progress reports showing the planned and actual effort for activities such as architectural reviews and rework. These also contain a number of metrics such as how many modules there were at each defined module status at the time of the progress report. One such metric was how many modules there were waiting for architectural review. The progress reports covered the construction phase with an interval of approximately one week.

Further, the primary author has drawn from experience related to participation in the project being investigated where he had the overall responsibility for work practices and tools. He has also drawn from experiences gained through participation in several other MDD projects, where he has either acted as an architect or as a mentor to the architect(s).

4 Integrating Architectural Design Rules in a MDD project

This section describes the research findings, beginning with a description of the research context.

4.1 Action Case Context

At the start of the project Combitech faced the challenging task of developing a software platform for a new generation of digital TV set top boxes for the DVB standard. The development had to be done in 18 months under strict quality
requirements and on a completely new, customized hardware platform developed in parallel by another company. At the time of the project, Combitech was a Swedish consultancy company specialising in services for developing embedded real-time systems. Combitech had approximately 250 consultants of whom about 75 where involved in the project. The total effort of the project was 100+ person years expended over a 24 month period, with the first delivery after 18 months and two more deliveries with additional functionality and corrections during the last six months. The project was distributed across five sites in the south of Sweden where the geographical distances between sites ranged between 1.5 – 4 hours one way travel by car. The two architects were stationed in the same site which meant that the other sites had to rely heavily on remote architectural support.

Combitech developed the platform as a phased fixed price (price negotiated for each phase) assignment for a customer company. Combitech had full responsibility for the software development but the customer wanted control of the architecture of the software so the architecture team was actually managed by the customer and not by Combitech, although they were Combitech employees. This meant that there was a need for a counterpart on the Combitech side, making sure that the architecture also was feasible within the budget. This was the project role of the first author of this paper: technically responsible within the Combitech project management team, with prime responsibility to negotiate the architecture with the architecture team on the customer side.

### 4.2 Project Challenges

The project faced a number of challenges:

- Short time to market: since Combitech were in competition with other developers the product had to be ready at a fixed date.
- Since the hardware was to be developed in parallel with the software there would be little time to integrate the two, leading to a significant risk of errors and misunderstandings that would have to be handled by very late changes in the software. Therefore Combitech needed to test the software on a range of platforms, from host PC to real target hardware with several intermediate hardware platforms.
The requirements were a moving target where the initial requirement specifications would be overridden by acceptance tests delivered late in the project.

The maintenance phase would be lengthy and had to be very cost effective.

There was a requirement to be able to differentiate the product, releasing different variants for different markets. New variants had to be developed and maintained efficiently.

Products would be competing on performance and quality; the product with the best performance and quality would win the final contract.

### 4.3 Rationale for choosing MDD

Combitech had experience from maintenance on the previous generation of the product which had been developed by another company. The product existed in many different variants for different markets so Combitech was confident that this would be the fact also for the new product. Combitech saw a potential to make the maintenance of the product a lot more efficient by building it according to a product line approach.

Within Combitech there was also extensive experience of working with models in UML and preceding modelling languages such as OMT, Booch, Coad-Yourdon and Objectory, both for analysis and design models. Combitech also had experience of using rule-based transformation from design models directly into code which executed on a platform. However, in real projects Combitech had so far only executed the transformations manually, although experimentation with automatic code generation had been done to a degree where the company felt ready to apply it in a real project.

Given this experience there was conviction amongst the project management team that model driven development would help address the challenges of the project by making the team more efficient, agile and flexible regarding the hardware platform:

- **Efficiency**: MDD would eliminate the manual and error prone step of implementing the UML models.

- **Agility**: The approach would make it possible to work in an agile way where one could quickly go from requirements to tested implementation without having to skip documentation, something very important for the maintainability of the product.
- **Flexibility in HW platforms:** MDD would make it possible to test most of the code without access to the actual hardware by simply generating code for different platforms, as the project gained access to hardware that became increasingly similar to the final target.

### 4.4 Tool selection

Given the tight deadline, an out-of-the-box tool solution was required that would give the following:

- Modelling in standard UML, to minimize the need for training.
- Generation of code with good performance on the target platform, the host platform and the platform for the previous generation of set-top boxes since this would be used as an intermediate test platform.
- 100% of the developed code generated from the model (to avoid synchronization problems with code and model) having at the same time the ability to use pure C++ code where there was a need (to eliminate a potential risk of not being able to do everything possible in the traditional way).
- The ability to debug at the model level.
- Support for distributed team working.
- A high probability that the vendor would continuously improve the tool towards the requirements of embedded real-time system development.

After a brief evaluation Rhapsody from Ilogix was selected as the only tool that seemed to satisfy all these requirements.

The selection of Rhapsody meant that the system was designed as a UML model with action code in C++. This model (the system model) was then automatically converted to full production code in C++ by the tool. The generated code uses an execution framework (OXF), provided by the tool, to abstract out the target execution platform. In terms of MDA [40], the model built in Rhapsody corresponds most closely to the PIM, where OXF and the generated code corresponds to the PSM.

### 5 The process

The project followed a phased process similar to the RUP model according to Fig. 4.
The main architectural work was done during the Elaboration phase during six months by the two architects. The Construction phase was started with a workshop where the architecture was presented to the design teams. The Construction phase was then performed by approximately 50 developers divided into seven teams during twelve months. To ensure that the design corresponded to the architecture the overall design of each component was reviewed by the architecture team before detailed design of the component was allowed to start. Since there were 166 components, this work occupied the architects almost full time during this phase.

5.1 Capturing the architecture

To be able to meet the deadline about 50 developers were assigned to the project after six months of architectural work undertaken by the architecture team. To be efficient they had to be able to work as independently of each other as possible. This required a stable architecture to be developed during these first six months before the project scaled up. A product line architecture approach [9] was selected to address the requirements for efficient development and maintenance of product variants. In addition to this there were other important quality requirements such as portability (it was anticipated that the software would outlive the hardware) and overall performance, which had to be handled by the architecture. So, an appropriate architecture was fundamental to the success of the project.
One of the first problems to face was how to represent the architecture when working in a MDD context. A basic idea in MDD is to use models instead of documents to represent the requirements and design of a system and to generate the implementation code from these models. The traditional way of representing the architecture is in a document that guides and constrains the detailed design. In model oriented processes, like Rational Unified Process (RUP) [25], where models have replaced requirement specifications and design descriptions, one still represents the architecture in a document. The aim of the project was, insofar as possible, to automatically connect the architecture to the design, thereby minimizing both the maintenance problem and the effort to enforce the architecture in the design. In the end the project management team settled for the following approach:

- The high level structure was to be captured in the system model as UML packages.
- Architectural design rules were to be captured in natural language in a text document supported by a UML class framework in the system model.
- Example components would be designed in a package in the system model illustrating how to follow the architectural rules.

5.1.1 High level structure

The high level partitioning of the system, down to a level at which individual components were to be developed by individual developers, was captured in a package hierarchy populated with classes acting as facades [15] for the actual components. The system was divided into a number of layers according to Fig. 5:

![Layer Diagram](image)

Fig. 5. The layers of the system
Hardware Abstraction Layer (HAL), delivered by the hardware manufacturer.

Board Support Package (BSP) delivered by the RTOS vendor.

RTOS (Real Time Operating System), this was the commercially available RTOS VxWorks.

RTOS Abstraction (OSAL), a wrapper for the RTOS delivered together with the modelling tool.

Common Media Drivers (CMD), a set of hardware driver components delivered by the project.

Common Media Platform (CMP), a set of components reusable on several hardware platforms and for several DVB (Digital Video Broadcasting) standards. This was developed in the project.

Proprietary API (PAPI), the adaptation layer for a specific DVB standard. These components were developed in the project.

Each component was modelled as a package that was to contain the classes that realized the component. Each of these packages was placed in a package representing one of the layers PAPI, CMP or CMD. The architects modelled the system down to the component packages then it was the job of the component designer to define the classes inside the component package.

5.1.2 Architectural design rules

A number of patterns and rules were to be followed when the components in the model were designed. To support these patterns and rules an architecture package with an executable framework was developed as a part of the system-model. Fig. 6 shows a UML class diagram, taken from the system model, showing the classes in this framework. The classes are described with the following text in the architecture document:
Fig. 6. Executable framework of the architecture

- **arcComponentRegistry**: The main task for the arcComponentRegistry is to be the resource provider for and of all arcComponents in a layer. The system’s layer dependencies are maintained via its access to other arcComponentRegistries. This is basically the dispatcher in a Client-Dispatcher-Server pattern, see [13].

- **arcComponentUser**: Being an arcComponentUser is the only way of retrieving another component. This is basically the client in a Client-Dispatcher-Server pattern, see [13].

- **arcComponent**: This is the basic logical architectural building block. This is basically the server in a Client-Dispatcher-Server pattern, see [13].

- **arcComponentFactory**: The arcComponentFactory implements the instantiation rules of arcComponents. There is at least one of these for each concrete specialization of arcComponent in the system. All arcComponentFactories are encapsulated by an arcComponentRegistry

- **arcProfile**: arcProfile is the “root” of the system configuration. Could be viewed as an instance of the reference architecture.

- **arcNotifyer**: The arcNotifyer notifies its attached arcListeners of important events. This is basically the publisher in a Publisher-Subscriber pattern, see [13].
− **arcListener**: The arcListener registers to an arcNotifier to become notified of important events. This is basically the subscriber in a Publisher-Subscriber pattern, see [13].
− **arcLocked**: This is a basic resource locking mechanism (semaphore-type). It is primarily intended for transaction locks on shared resources.
− **arcLockedUser**: This is the interface required by an arcLocked component.
− **arcPort**: An arcPort is a system resource locked to one user at a time. The main purpose of a port is to transmit data to and/or from its user.
− **arcPortUser**: This is the interface required by an arcPort component.

In principle, the framework contained abstract base classes representing the core abstractions of the system, relations between them and operations that were to be overridden in specializations of the base classes. The framework also contained full implementations of some basic mechanisms that operated solely on the level of the abstract base classes, such as inter-process resource locking and component registry handling (registration, allocation and de-allocation).

Unfortunately the project could not fully capture the architecture in a formal model. It was necessary to use natural language to express the rules on how to use the architectural framework to design the components in the architecture. There were more than sixty rules that had to be followed. Below is a small, representative subset of these rules taken directly from the architecture rules document.

− “Any arcComponent with behaviour similar to an arcPort should be a specialization of arcPort.”
− “All specializations of arcPort may have one overloaded method for each of the methods Open(), Close() and Write()”.
− “All specializations of arcPort may have several methods for the method Ctrl(). These methods shall be named as ctrl_\_<specific_name> and may not change the parameter list of the base class, except for specialization of the parameter classes given for the base class. However, a method may omit the second (parameters) parameter.”
− “arcPort::Write() shall be used to stream data to a port’s data output stream.”
− “arcPort::Ctrl() shall be used to control and manipulate a port’s properties.”
− “All specializations of arcPort must use its parent’s implementation of the base class method for their respective purposes.”

89
- “All specializations of arcPort require a specialization of the arcPortUser interface.”
- “All specializations of the arcPortUser interface base class may have one overloaded method for each of the methods RxReady(), TxDone() and GetMem().”
- “All specializations of the arcPortUser interface base class shall have one overloaded CtrlAck() method for each of the asynchronous ctrl_<name> methods.”

5.1.3 Providing example components

To guide the developers in how to develop the components using the architectural framework, a couple of example components were also developed by the architects as a part of the model:
- A component showing how to realize the “pipes and filter” pattern.
- A component showing how to use interrupts.
- A component showing how to design a “port” component (a specialization of the arcPort component). This example was however never completed.

In addition to showing the design, the examples also showed how to use different diagrams to capture the design.

5.2 Manual Reviews of Architectural Conformance

Using natural language to describe architectural design rules meant that the project had to rely heavily on manual reviews to enforce conformance with the architecture. Performing these reviews was a heavy burden for the architects; it took almost all of their time during the first 12 months of development after the first release of the architecture.

The rules proved to be ambiguous and hard to comprehend, and thus prone to different interpretations. Several developers reported having a hard time trying to understand and follow all of the detailed rules. This was manifested by the fact that major corrections were frequently needed after reviews, as shown by the review protocols. Sometimes this meant that a lot of reworking had to be done since reviews were often held when design had continued too long. This was caused by work overload on the architects, which in turn was caused by a lot of effort expended on reviewing the designs generated by the
different teams. The progress reports show that it was common that architectural reviews were actually done after a component was completed and tested, which was in violation of the rules which stated that the component had to pass the review before the detailed design was done.

6 Conclusions

Architectural design rules are an important part of the architecture and there are no suggestions on how to model them in the current body of literature. The inability to formalize the architectural design rules leads to a need for manual enforcement of them. The research presented here shows that this is an error-prone and time-consuming task that takes most of the effort of the architects during the construction-intensive phases of a project. This problem exists in traditional document-based development as well as in MDD but it is more apparent and acute in MDD. This is because MDD has been able to automate the step from detailed design to implementation eliminating time consuming coding and code reviews. However, MDD has not been able to automate enforcement of the architecture on the detailed design due to the inability to model architectural design rules. The presented case shows that the inability to model architectural design rules makes architectural enforcement a bottleneck in MDD projects. This leads to a plethora of problems, including:

- **Stalled detailed design:** The design teams have to wait for the architects to review their overall design before they can dig deeper into the design.
- **Premature detailed design:** Design teams start detailing their design before their overall design is approved by the architect, with the risk that they will have to redo much work after the review.
- **Low review quality:** Low quality of the reviews, leading to problems later in the project.
- **Poor communication of the architecture:** The architects have no time to handle the communication with the design teams regarding architectural interpretations or problems, problems are “swept under the carpet.”

The implications for theory are that there is a need for further research to find ways of modelling architectural design rules in such a way that they are both amenable to automatic enforcement on the detailed design and easy to understand and use by both
architects and developers. The implications for practice are that, until there is support for automatic enforcement of architectural design rules, extra resources are needed for architectural reviews. This has to be taken into account in the planning and in the architectural work practices. Although the architectural design is inherently a task for a relatively small group, it should be possible to delegate the architectural reviews to a larger group. This would give time for the architects to concentrate on the core architectural tasks; designing, communicating and maintaining the architecture.

Acknowledgments

This research has been financially supported by the ITEA project COSI (Co-development using inner & Open source in Software Intensive products) (www.iteacos.org) through Vinnova (www.vinnova.se) and also by funding from Science Foundation Ireland (www.sfi.ie).

References


An Approach for Modelling Architectural Design Rules in UML and its Application to Embedded Software

Anders Mattsson¹, Brian Fitzgerald², Björn Lundell³, Brian Lings³

1. Combitech AB, Jönköping, Sweden
2. University of Limerick, Limerick, Ireland
3. University of Skövde, Skövde, Sweden

ACM Transactions on Software Engineering and Methodology, vol. 21, issue 2, 2012

Research and writing: Anders Mattsson
Critical revision: Brian Fitzgerald, Björn Lundell, Brian Lings
Abstract

Current techniques for modelling software architecture do not provide sufficient support for modelling of architectural design rules. This is a problem in the context of Model-Driven Development in which it is assumed that major design artefacts are represented as formal or semi-formal models. This paper addresses this problem by presenting an approach to modelling architectural design rules in UML at the abstraction level of the meaning of the rules. The high abstraction level and the use of UML make the rules both amenable to automation and easy to understand for both architects and developers, which is crucial to deployment in an organization. To provide a proof-of-concept a tool was developed that validates a system model against the architectural rules in a separate UML model. To demonstrate the feasibility of the approach the architectural design rules of an existing live industrial strength system were modelled according to the approach.
1 Introduction

A basic premise of Model-Driven Development (MDD) [Schmidt 2006] is to capture all important design information in a set of formal or semi-formal models that are automatically kept consistent by tools. The purpose is to raise the level of abstraction at which the developers work and to eliminate time consuming and error-prone manual work in maintaining consistency between different design artefacts such as UML diagrams and code. An important design artefact in any software development project is the software architecture [Bass et al. 2003]. The purpose of the architecture is to guide and control the design of the system so that it meets its quality requirements. A common way of capturing the architecture in MDD projects is to put the high level structure in the form of packages and components with interfaces in the system model, together with a framework implementing a communication infrastructure used by the components [Mattsson et al. 2009]. This is however not enough; you also need to specify rules as to what kinds of component to put in different layers and how these are supposed to use the infrastructure. We call these rules architectural design rules [Mattsson et al. 2009]. The current state of practice is to express these rules in informal text for the developers to follow. This means that manual reviews have to be used to check that the rules have been followed during detailed design. If we could model architectural design rules in a form that could be interpreted by tools, and at the same time easily be understood by both architects and developers, we would be able to eliminate error prone and time consuming manual work.

In this paper we present an approach to solving this problem by using the well-known modelling language UML [OMG 2009] to define architectural design rules at the metamodel level in an intuitive way. To verify that the approach can be automated, a tool has been built that checks that a system model conforms to architectural design rules modelled according to the approach. To demonstrate the applicability of the approach to real systems development the architectural design rules of an already developed real-world embedded system have been modelled according to the approach.

The rest of this paper is organized as follows. In the next section we present the background motivating the research. In section three we introduce a fictional but realistic example to illustrate the problem of modelling architectural design rules and to introduce our proposed solution. Thereafter we present the research approach adopted.
for the study. Following this, our findings are presented in three consecutive sections covering the definition of the approach, tool support for automation, and the results from applying the approach to a real-world system. Finally we discuss our conclusions and the implications of the findings.

2 Background

Our main research objective was to define an approach for modelling architectural design rules in an intuitive way while also stringent enough for automation. In order to motivate and validate our research objective we conducted a literature review in line with [Levy and Ellis 2006]. The review consisted of two consecutive phases where the first phase focused on the role of architectural design rules in the context of MDD. The findings from this phase are presented in section 2.1. Since we have reported these findings in [Mattsson et al. 2009], where a detailed discussion can be found, this part is kept brief in this paper. Informed by the findings of the first phase the second phase of the literature review focused on techniques for using UML to constrain UML modelling. The findings from this phase are presented in section 2.2.

2.1 Architectural Design Rules and MDD

The purpose of the architecture is to guide and control the design of the system so that it meets its quality requirements. Bass et al. [Bass et al. 2003] are unequivocal in stating the importance of an architectural approach:

“The architecture serves as the blueprint for both the system and the project developing it. It defines the work assignments that must be carried out by design and implementation teams and it is the primary carrier of system qualities such as performance, modifiability, and security – none of which can be achieved without a unifying architectural vision. Architecture is an artefact for early analysis to make sure that the design approach will yield an acceptable system. Moreover, architecture holds the key to post-deployment system understanding, maintenance, and mining efforts. In short, architecture is the conceptual glue that holds every phase of the project together for all of its many stakeholders.”

A common understanding in architectural methods is that the architecture is represented as a set of components related to each other [Perry and Wolf 1992; Shaw et al. 1995].
The components can be organized into different views focusing on different aspects of the system. Different methods propose different views; there may be a view showing the development structure (e.g. packages and classes), a view showing the runtime structure (processes and objects) and a view showing the resource usage (processors and devices). In any view each component is specified with the following:

- an interface that documents how the component interacts with its environment.
- constraints and rules that have to be fulfilled in the design of the component.
- allocated functionality.
- allocated requirements for quality attributes.

A typical method of decomposition (see for instance [Bass et al. 2003], [Wojcik et al. 2006] and [Bosch 2000]) is to select and combine a number of patterns that address the quality requirements of the system and use them to divide the functionality in the system into a number of elements. Child elements are recursively decomposed in the same way down to a level where no more decomposition is needed, as judged by the architect. The elements are then handed over to the designers who detail them to a level where they can be implemented. For common architectural patterns such as Model-View-Controller, Blackboard or Layers [Buschmann 1996] this typically means that you decompose your system into subsystems containing different kinds of classes (such as models, views and controllers). However the instantiation into actual classes is often left to the detailed design, for two main reasons:

1. Functionality will be added later, either because it was missed or because a new version of the system is developed, so more elements will be added later that also have to follow the design patterns decided by the architect.
2. It is not of architectural concern. The concern of the architect is that the design follows the selected architectural patterns, not to do the detailed design.

This means that a substantial part of the architecture consists of design rules as to what kinds of elements, including behavioural and structural rules and constraints, should be in a certain subsystem.

The importance of architectural design rules is also highlighted in current research in software architecture which is focused on the treatment of architectural design decisions.
as first class entities [Jansen and Bosch 2005; Jansen et al. 2007; Kruchten 2004; Kruchten et al. 2006; Tyree and Akerman 2005], where architectural design decisions impose rules and constraints on the design together with a rationale. However, there is not yet any suggestion on how to formally model these design rules. The current suggestion is to capture them in text and to link them to the resulting design. This may be sufficient for rules stating the existence of elements (“ontocrisis” in [Kruchten 2004]) in the design, such as a subsystem or an interface, since the architect can put the actual element (i.e. a certain subsystem) into the system model at the time of the decision. It is however not sufficient for rules on potentially existing elements (“diacrisis” in [Kruchten 2004]) such as rules as to what kinds of elements, including behavioural and structural rules and constraints, should be in a certain subsystem, since the actual elements are not known when the design decision is made. Instead, the rule-based design occurs later in the detailed design phase, and involves other persons, potentially even in a different version of the system.

As previously reported [Mattsson et al. 2009] there is no satisfactory solution to how to model architectural design rules on potentially existing components in the current body of literature:

- Approaches to MDD, such as OMG’s MDA [OMG 2003], Domain Specific Modelling (DSM) [Karsai et al. 2003; Tolvanen and Kelly 2005], and Software Factories [Greenfield and Short 2004] from Microsoft do not address the problem of how to represent architectural design rules.

- Numerous methods exist for architectural design such as ADD[Bass et al. 2003; Wojcik et al. 2006], RUP 4+1 Views [Kruchten 2004; Kruchten 1995], QASAR [Bengtsson and Bosch 1998; Bosch 2000; Bosch and Molin 1999], S4V [Hofmeister et al. 2000; Soni et al. 1995], BAPO/CAFCR [America et al. 2004; van der Linden et al. 2004] and ASC [Ran 2000]. Also, current research in software architecture is focused on treating architectural design decisions as first class entities [Jansen and Bosch 2005; Jansen et al. 2007; Kruchten 2004; Kruchten et al. 2006; Tyree and Akerman 2005]. However neither of these research streams provides any suggestion as to how architectural design rules should be modelled, other than as informal text.

- Architectural Description Languages (ADL) [Medvidovic et al. 2007; Medvidovic et al. 2002; Medvidovic and Taylor 2000] (e.g. ACME, Aesop, C2,
MeatH, AADL, SysML and UML) do not provide sufficient means to specify constraints or rules on groups of conceptual components only partly specified by the architect where the actual components are intended to be indentified and designed by developers in later design phases.

The state of the art in embedded software development [Mattsson et al. 2009] is to capture these rules in a text document. This means that we have to rely on manual reviews to ensure that the detailed design follows the architectural design rules. As a consequence, architectural enforcement becomes a bottleneck in MDD, where other design activities have been automated. As earlier reported [Mattsson et al. 2009] this leads to a plethora of problems, including:

1. **Stalled detailed design:** The design teams have to wait for the architects to review their overall design before they can dig deeper into the design.

2. **Premature detailed design:** Design teams commence detailed design before their overall design is approved by the architect, with the risk that they will have to redo much work after the review.

3. **Low review quality:** Time pressures lead to a low quality of review, leading to problems later in the project.

4. **Poor communication of architecture:** The architects have no time to handle the communication with the design teams regarding architectural interpretations or problems; problems are “swept under the carpet.”

An architectural style (also known as architectural pattern) [Shaw and Garlan 1996] is an idiomatic pattern of system organization. It is comparable to the solution part of a certain kind of design pattern [Gamma 1995] specifying system wide design rules, categorized as architectural patterns in [Buschmann 1996].

The problem of modelling design rules has much in common with the problem of modelling architectural styles or the solution part of a design pattern in so far as it basically is about specifying rules to follow in the design. There are a number of suggestions on how to formally model design pattern specifications and architectural styles [Bayley 2007; Eden 2002; France et al. 2004; Lauder and Kent 1998; Mak et al. 2004; Mikkonen 1998; Pahl et al. 2007; Zdun and Avgeriou 2005]. While some approaches use mathematical formalisms such as predicate logic and set theory, others
use UML applied at the meta-model level. Based on our experience we believe that, in order to be successful in practice, it is essential that architectural design rules are modelled in such a way that they are both amenable to automatic enforcement of the detailed design and easy to understand and use by both architects and developers. The latter is important in order to avoid increasing the work of developing the rules; otherwise there is a risk that the work burden is increased instead of decreased even though the enforcement is automated. Another important issue is that it should be possible to use current mainstream modelling tools to model both the architectural design rules and the system model so as to make it widely adoptable. Given that UML is probably the most widely used modelling language in the embedded software industry our choice would therefore be to use UML to model architectural design rules for UML models. Our approach is therefore based on the same idea as in [Zdun and Avgeriou 2005] and [France et al. 2004], namely to use UML on the meta-model level to restrict the use of UML in a system model. However, instead of using it to specify patterns, we use it to specify architectural design rules.

### 2.2 Architectural Design Rules in an MDD Context Using UML

The purpose of architectural design rules is to provide the necessary constraints for the detailed design. In an MDD context where the detailed design is made in UML this means that the architectural design rules must be modelled in such a way that they restrict how UML is used. Furthermore, to suit our purpose, it must be possible to automatically enforce the restrictions on the detailed design or to automatically check that the restrictions are followed in the detailed design. Within UML, a profile provides a mechanism to restrict the use of UML [Fuentes-Fernández and Vallecillo-Moreno 2004]. A UML profile contains a number of stereotypes where each stereotype extends one or more UML meta-classes with new properties and constraints. The stereotype can then be applied to model elements of the extended meta-class in a model using the profile. In Fig. 1 an example is given where we define a stereotype Data_Class that extends the UML meta-class Class. The stereotype adds the constraint that Classes with the stereotype Data_Class cannot have any operations. The constraint is expressed in OCL [OMG 2003], a language for specifying constraints and queries on models in UML and other MOF [OMG 2006] based languages defined by OMG (MOF is a subset of UML intended for meta-modelling). The application of this stereotype is shown in Fig. 2 where we define a class Position with the stereotype Data_Class.
There are, however, at least two problems with defining profiles in this way. The first problem is that it requires detailed knowledge of the UML meta-model (the model defining the abstract syntax of UML), which is quite complex and beyond what can be expected from a typical architect or developer; it would be likely to impede widespread adoption of the approach [Conboy and Fitzgerald 2010]. For example, the very simple constraint in Fig. 1 requires the knowledge that operation has the role name `ownedOperation` in its association to `Class` in the UML meta-model. The second problem is that the OCL expressions become quite complex even for quite simple constraints. Consider the following example rule (rule S4 in the illustrating example in section 3):

\[ \text{A sensor may only have associations to } \text{In\_Port\_Ifc and Data\_Items. These associations shall only be navigable from the sensor.} \]

Using the standard approach for defining profiles we get the constraint definition for the Simulator stereotype shown in Fig. 3. As can be seen, this involves a great deal of detailed knowledge of the UML meta-model.
Another possibility is to make a new meta-model with classes that extend the classes in the UML meta-model through generalizations. But, as illustrated in Fig. 4, this is very similar to the approach using a profile, in that one still needs to specify almost the same OCL constraints as when using a profile. The only benefit is that you avoid the navigation between the stereotypes and the elements in the meta-model (e.g. self.base_Class and type.extension_Data_Type)

![Diagram](image)

Fig. 3 Definition of the architectural design rule example using OCL

What is needed is a technique to specify the constraints in a more intuitive way. In [Fuentes-Fernández and Vallecillo-Moreno 2004] a technique using a meta-model as a
precursor to a UML profile specification is suggested. According to this approach stereotypes are defined by classes in a meta-model where the relations between the classes impose constraints on the stereotypes. Using an approach like this the above example would be expressed according to Fig. 5.

Fig. 5 Capturing the architectural design rule example using the approach suggested in [Fuentes-Fernández and Vallecillo-Moreno 2004]

This approach has the benefit that it is more intuitive, it is both easier to model and to understand. Another benefit is that it does not contain any details from the UML meta-model so it does not require any knowledge of that. The drawback of this approach is that it lacks rigour on how to transform it to a UML profile. In the context addressed in [Fuentes-Fernández and Vallecillo-Moreno 2004] this is not a problem since the purpose of the model is just to aid in the process of designing a profile, not to be automatically transformed into a profile. For our purpose a detailed specification as to how it may be transformed to a UML profile is necessary, to the level where it could be implemented in a tool. To that purpose we have defined a set of transformation rules. These are described in section 5.

3 An Illustrative Example

In this section we introduce a fictional but realistic example to illustrate the problem of modelling architectural design rules and to introduce our proposed solution to that problem.

Our fictional system is a product line of washing machines. The product line consists of a wide variety of washing machines from simple cheap machines with a minimum of features to advanced machines with user access control that are monitored and controlled over the internet for industry and public self-service laundries. Since there is a high degree of functional commonality between different machines it has been decided to build a common model from which software for all machines (existing and future) can be generated. With this goal there are a number of non-functional requirements that must be addressed by the architectural design, such as:

1. Performance scalability: In simple machines it shall be possible to run the software in a microcontroller with very limited performance and memory while
the more advanced machines have fully featured CPU’s with hundreds of megabytes of memory.

2. **IO hardware variability:** Since the availability and price of IO hardware varies over time change of IO hardware shall require minimal effort.

3. **Communication protocols variability:** Since different machines use different protocols for communication with external systems now and in the future, change of communication protocols shall require minimal effort.

4. **Functional scalability:** Since the functionality is highly variable adding, removing and changing functionality, including beyond what is considered currently, shall require minimal effort.

5. **User interface variability:** There is high variability in how the user controls the machines, from simple variants with knobs and LED’s to touch screens for the most advanced machines. Therefore, it shall require minimal effort to change the interface to the user, including beyond the controls existing currently.

6. **Sensor and actuator variability:** While some machines use actual sensors to monitor water temperature and water level others use time to estimate these values. There are also different scalings between the sensor output and measured values for different physical sensors and different machines (e.g. for water level). Depending on the functionality of the machine there are also different kinds of sensor and actuator for different machines (e.g. if the machine also has tumble drying functionality or dirt sensing capabilities). To cater for this variability adding removing or changing sensors and actuators shall require minimal effort.

To handle these requirements the following design principles have been decided by the architect:

1. **Performance scalability:** No heavy-weight functionality is required. For example, it might have been sensible to use a database with remote accessibility to store the data items since this would have eliminated the need for implementing support for remote accessibility. However, this would have made it impossible to run the software on a microcontroller.
2. **IO hardware variability:** The IO hardware is only accessible through a small stable set of IO interfaces. These interfaces are then implemented for the different hardware by different IO_Ports.

3. **Communication protocol variability:** This is handled by the same design principle as used for handling the IO hardware variability. Different protocols are handled by different IO_Ports towards the same stable interface.

4. **Functional scalability:** This is handled by not allowing any dependencies on or between applications (e.g. washing program, remote monitoring, access control…). An application reads and writes to Data_Items and IO_Interfaces and may act as an observer [Gamma 1995] on Data_Items reacting to changes on these.

5. **User interface variability:** This is handled by using the same principles for user interface controllers as for the applications described above. A user interface controller provides a mapping between a physical user interface and Data_Items.

6. **Sensor and actuator variability:** This is handled by using the same principles for sensors and actuators as for the applications described above. Sensors and actuators provide a mapping between physical sensors or actuators and Data_Items.

In the following two subsections we first show how an architecture capturing these design principles would have been modelled in a traditional way and then using our modelling approach.

### 3.1 Traditional Way of Modelling the Architecture

Traditionally the architect would have documented the architecture according to these design principles with a high level structure and a support framework in UML together with a set of rules expressed informally in text. This is exemplified with the UML model in Fig. 6 accompanied by a set of textual architectural design rules, such as the ones below the figure.
Some rules for Data_Items:

D1. A Data_Item is a class that reflects the state of the system or its environment that is needed by an application. The intention is that the set of Data_Items shall be stable over time.

D2. A Data_Item shall inherit Infrastructure::Subject.

D3. A Data_Item shall be defined in the Data_Items package.

D4. The only public operations of a Data_Item shall be set and get operations to read and write data stored by the class.

D5. A Data_Item may be a composition of Data_Items.

D6. A set operation for a Data_Item shall always end by calling its Notify operation.

Some rules for sensors:

S1. A sensor is typically responsible for reading the value from a physical sensor scaling it and writing the value to a Data_Item. Some sensors may however not
be connected to a physical sensor but use indirect measures such as heating effect and time to estimate a value to write to the Data_Item.

S2. A Sensor shall be defined in the Sensors package.

S3. A sensor may inherit Infrastructure::Observer to be able to react to changes in Data_Items, for instance to activate or deactivate itself.

S4. A sensor may only have associations to In_Port_Ifc and Data_Items. These associations shall only be navigable from the sensor.

S5. A sensor may not have any public operations or attributes.

S6. A sensor shall update its Data_Item periodically.

In addition there would be corresponding rules for Actuators, UIControllers, Applications and IO_Ports.

3.2 Modelling the Architecture According to Our Approach

In our approach, instead of using informal text, the architectural design rules are modelled in UML. Since UML (and any other OO language) is well suited to define structural relationships (such as for instance “every country has one capital city” used in many introductory courses in OO), this can be done in a straightforward way for rules such as the ones in the previous section. Using this approach the architectural rules for Data_Items above can be modelled as exemplified in Fig. 7. In the figure it is indicated how each rule is modelled by the Dx labels. For example the rule D2, stating that a Data_Item shall inherit Subject, is modelled by associating a Data_Item to one Subject with an association stereotyped with <<Generalization>>. A major principle is that nothing that is not explicitly allowed is forbidden so a Data_Item may not have any association other than compositions to other Data_Items and may not inherit anything except a subject class.

We call the model where we model the architectural rules the architectural rules model. It is important to realize that the classes in this model are at the meta-level of the classes of the system model; that is, they define different kinds of classes and constrain them. For instance, the association between Subject and Observer in Fig. 7 means that a Subject kind of class shall have any number of Observer kind of classes. An operation or an attribute in a class in this model means that a class of the corresponding kind in the system model must have an operation or attribute with the same characteristics as
this operation or attribute. To allow for variations, wild cards can be used in attribute and operation definitions, where “@” or “%” stands for any character sequence and “%” has the additional meaning that an element with “%” in its name may be repeated any number of times, including zero. A full definition of the constructs of the architectural rules model is presented in section 5.

In the system model we use UML stereotypes to show the kind, corresponding to the classes in the architectural rules model, of an element. For example, in Fig. 8 it can be seen that the class Subject has the stereotype <<Subject>> meaning that it has to comply with the constraints defined by the Subject class in the architectural rules model. Fig. 8 shows the high-level structure and framework classes modelled in the system model by the architect. The only difference to the one in Fig. 6 modelled according to traditional approach is the stereotypes attached to the packages and classes. Since the architect models the high-level structure of the system, the rules restrict the developers as to which stereotypes to use in which package. For instance, in a package with stereotype <<Data_Items>> all classes must have the stereotype <<Data_Item>>. Fig. 9 shows a number of Data_Items modelled in the system model following the rules.
of the architectural rules model in Fig. 7. Any violations to the rules are automatically detected and reported by the tool built as part of our case study, described in section 6.
An interesting observation in this example is that rule D1 is not modelled. The reason is that it is too vague to be formalized; it requires the developer to exercise judgement to follow it. In our experience there are these kinds of architectural rules in most systems so 100% automation of the architectural reviews is not a realistic goal. Nevertheless, we should be able to model and automatically enforce a majority of the architectural rules, something which should have a major impact on development efficiency.

4 Research Approach

There were three objectives of this research motivated by the lack of a satisfactory solution on how to model architectural design rules and the need for automation of the enforcement of these in a practical situation as discussed in section 2:

1. Define an approach for modelling architectural design rules in UML
2. Verify that the approach is stringent enough to be automated
3. Demonstrate that the approach is applicable to a real development project

To achieve the first objective, that of defining the approach, a systematic literature review presented in section 2 was performed. The approach adopted was based on the findings of the literature review, and was refined based on the activities undertaken to achieve the second and third objectives above.

The second objective, to verify that the approach could be automated, was addressed by developing a tool to automatically check that a system fulfilled the architectural rules specified according to the approach. In addition a MOFScript transformation that transforms architectural rules defined according to our approach to OCL constraints in an architectural rules profile has been defined. This MOFScript transformation can be found in appendix A.

To achieve the third objective, that of demonstrating the applicability of the approach, the architectural design rules from an existing, previously developed system, were modelled according to the approach. The goal of this activity was to establish the degree to which it enabled modelling of the architectural design rules. Specifically, we searched for answers to the following two questions:

1. To what extent could the specified rules be modelled?
2. Were there certain kinds of rule that could not be modelled and if not, why not?
The system was selected based on the following criteria:

1. The system had to have been developed using MDD.
2. The system had to be an existing real system of significant size and with a sufficient functionality to make it generally representative as a real-world embedded system.
3. The architecture, including the architectural design rules, had to be documented to a level where it could be interpreted by the research team.
4. The research team had to have good access to people who had first-hand knowledge of the architecture, to be able to see beyond the documentation and to be able to resolve any ambiguities.

The selected system, fulfilling these criteria, was a software platform for digital TV set top boxes for the DVB\(^4\) standard. The system had been developed by a project which had been studied in an earlier case study by the research team, reported in [Mattsson et al. 2009], which meant that the team had good insight into the case. It was developed using the modelling tool Rhapsody (version 4.x) from Telelogic [Telelogic], with all code generated from UML models in the tool, using C++ as the action code language. The size of the software platform was approximately 350,000 eLOC in C++ and the effort to develop it was about 100 person years over a 24 month period. The architecture was documented partly in the system model and partly in one manually written document. The system model contained a high-level package structure and a framework of classes supporting the architectural design rules. The document contained the architectural design rules. Finally, the researchers had first-hand knowledge of the architecture since the primary author of this paper was the technical manager of the project, responsible for work practices and tools. The architecture was however developed by two other persons acting as architects.

The study was conducted by a systematic walkthrough reviewing the rules from the architectural document in several iterations, gradually transforming them to modelling constructs according to our approach.

---

5 An Approach to Modelling Architectural Design Rules

In this section we present the definition of the approach for modelling architectural design rules that was developed in response to our first research objective.

As motivated in section 4 our approach is based on transforming design rules modelled in an architectural rules model, using UML, to a UML profile that are applied to the system model. The implication is that our approach to modelling architectural design rules can be reduced to a set of transformations from constructs in the architectural rules model to stereotypes with constraints in a UML profile, hereafter referred to as the architectural rules profile. Therefore our approach is defined using such a set of transformations. In this section we present these transformations in an informal descriptive way, a formal definition in the form of a MOFScript transformation to a UML profile and OCL constraints can be found in appendix A.

The transformations are divided into two subsets, a general, complete transformation set and an additional UML specific transformation set. The first transformation set is general in the sense that it is applicable to any meta-model modelled in MOF, not only UML models. It is also complete in the sense that it allows us to constrain any construction of any modelling language defined in MOF. However, using only these general transformations it is still hard to model certain types of architectural rules commonly needed for UML models, for example rules restricting UML associations. To ease the modelling of such rules, the additional UML specific set of rules is needed. This transformation set is, however, not complete so the fundamental set is still needed for completeness.

All examples illustrating the transformations in this section are taken from the washing machine example introduced in section 3.

5.1 General Transformations

This section defines a set of transformations from an architectural rules model to an architectural rules profile defining constraints for types of classes in a system model. The transformations are applicable to all MOF based languages, not just UML. The definitions refer to the generic architectural rules model in Fig. 10 where C1 and C2 are replaced with class names, M1 and M2 replaced with stereotypes, R1 and R2 are replaced with role names, SR1 and SR2 are replaced with stereotypes and Mu1 and
Mu2 are replaced with multiplicities. In the transformations below the following conventions are used:

- References to terms defined in the generic architecture model in Fig. 10 are in italics.
- The phrase “<<Cx>> element” shall be interpreted “element of stereotype Cx” or, if Mx equals “meta-class”, “element of meta-class Cx” (where x is 1 or 2).
- The term “meta-class” in the transformations refers to a meta-class of the modelling language that is constrained, for instance the meta-model for UML.
- The term “meta-model” in the transformations refers to the meta-model of the modelling language that is constrained, for instance the meta-model for UML.

![Diagram](image)

Fig. 10 A generic architectural rules model used in the definition of the transformations

The transformations are:

**T1.** A class named C1 with the stereotype M1 is transformed into a stereotype named C1 extending the meta-class M1 unless transformation number T2 below applies. If M1 is undefined then “Class” is assumed; see Fig. 11 for an example.

![Diagram](image)

Fig. 11 Example of transformation according to transformation T1.

**T2.** If M1 equals “metaclass” then C1 represents the class C1 in the language meta-model (i.e. the UML meta-model) and is not transformed into anything in the
profile. This can be used to specify constraints in other stereotypes in respect to these meta-classes, see Fig. 12 for an example.

T3. If SR2 is the role in the language meta-model on the far end of an association from the meta-class of C1 to the meta-class of C2 then the multiplicity of R2 for a <<C2>> element shall be constrained to Mu2 in stereotype <<C1>>.

An example is shown in Fig. 12 where a <<Sensor>> class is constrained to only have one <<SamplingPeriod>> attribute and no other attributes.

It is allowed to have several association ends matching the same meta-model association end. In that case the multiplicity of the end with the most narrow type scope is applied for a certain <<C2>> element. In the example in Fig. 12 for example the multiplicity is “1” for an attribute with the stereotype <<SamplingPeriod>>, since this multiplicity is only applicable to attributes with stereotype <<SamplingPeriod>> and the multiplicity of 0 is applicable to all attributes.

**Architectural rules model**

```
<<Class>>
Sensor
```

```
<<Property>>
SamplingPeriod
name = "SamplingPeriod"
visibility = "private"
```

**Architectural rules profile**

```
<<Stereotype>>
Sensor
[A <<Sensor>> Class must have one ownedAttribute of the stereotype SamplingPeriod an no other ownedAttributes]
```

```
<<Stereotype>>
SamplingPeriod
[A <<SamplingPeriod>> Property must have the name "SamplingPeriod" and the visibility 'private']
```

Fig. 12 An example of using transformation T2, T3 and T4.

T4. If the name of an attribute A matches the name of an attribute of class M1 in the meta-model then it is transformed into a constraint on that attribute on allowed values. The value of the attribute is constrained to match a regular expression specified as the default value of the attribute.

An example is shown for the attribute name in Fig. 12 where the name of the <<SamplingPeriod>> attribute is constrained to be “SamplingPeriod”.

T5. If no match is found for A then A is transformed into an attribute A of the stereotype (tag-definition), thus defining a tagged value to be set in the model element where the stereotype is applied.
T6. Any OCL constraint in the context of a class \( C1 \) is copied into the architectural rules profile with the context of stereotype \( C1 \). This means that the constraints shall be written the same way as when defining stereotypes directly in the profile.

Even though OCL expressions, as discussed in section 3, are not suitable for modelling architectural design rules in general, there is a need for them to express for instance constraints on combinations of rules. For example, if we would like to specify that a <<Sensor>> class has either a <<Sample>> operation or a <<Trig>> operation it could be done like this:

```plaintext
context Sensor
inv: self.base_Class.ownedOperation.extension_Sample.size()=1 xor
    self.base_Class.ownedOperation.extension_Trig.size()=1
```

T7. A generalisation relationship from a class \( C3 \) to a class \( C1 \) in the architectural rules model is transformed to a generalisation from stereotype <<C3>> to stereotype <<C1>> in the architectural rules profile as exemplified in Fig. 13. This means that stereotype <<C3>> inherits all constraints from stereotype <<C1>> and that a <<C3>> class is also to be regarded as a <<C1>> class.

---

This set of transformations is general and complete in the sense discussed below:

- **The transformation set is general:** These transformations allow us to use a sub-set of UML to constrain the usage of any modelling language defined in MOF since the transformations only assume that the modelling language is defined using MOF and do not assume anything about the content in the meta-model (i.e. the UML meta-model).

- **The transformation set is complete:** A model is an instance of its meta-model, which means that any model element is an instance of a class in the meta-model. The only things that may vary between two models of the same meta-model...
defined in MOF is the number of instances of each meta-class, the values and multiplicities of the meta-class attributes and the links between the instances. Since these transformations allow us to constrain allowed values and multiplicities for attributes and constrain the types and multiplicities of associations, the set of transformations is complete in the sense that it allows us to constrain anything that can vary between different models of a certain meta-model defined in MOF.

By only using these transformations it is, however, still too complex to model constraints on some common UML constructs such as associations, attributes, operations and state machines. For example, Fig. 14 shows how the simple example rule S4 introduced in section 3 is modelled according to these transformations.

![Fig. 14 Capturing an association constraint in a meta-model using the general transformations.](image)

To overcome this problem we have defined a set of additional transformations that makes it considerably simpler to specify certain constraints on UML models, common within the embedded software domain; these are described in the next section.

### 5.2 Additional UML Specific Transformations

This section defines a set of transformations in addition to the general ones defined in the previous section. The purpose of these transformations is to make it simpler to describe frequently needed architectural rules on UML models that are hard to describe.
using only the general transformations. These transformations override the general transformations in cases where both a general and an additional UML specific transformation apply. The definitions refer to the generic architectural rules model in Fig. 15. In the definitions the following conventions are used:

- References to terms defined in the generic architecture model in Fig. 15 are in italics.
- The phrase “<<Cx>> element” where x is 1 or 2 shall be interpreted “element of stereotype Cx” or, if Mx equals “metaclass”, “element of meta-class Cx”.

![Diagram](image1)

Fig. 15 A generic architectural rules model used in the definitions of the transformations

Constraints on stereotype C1 is defined according to the following:

T8. If M1 equals “Package” and aggregation of R2 is “composite”:

A <<C1>> package is constrained to contain Mu2 number of <<C2>> elements. The visibility of these elements shall be the visibility of Mu2. Also, a <<C1>> package is not allowed to have any packagedElements unless explicitly allowed in the model. This transformation makes it easy to model rules on package containment. An example is shown in Fig. 16.

![Diagram](image2)

Fig. 16 Example of rules on package containment.

T9. <<C1>> elements are only allowed to have the associations, dependencies, generalizations and realizations explicitly allowed.

T10. If MA equals “Association”:
A \(<\text{C}1\)> element shall be associated with \(\text{Mu2}\) number of \(<\text{C}2\)> elements. The association ends shall have the same navigability, aggregation (none, shared or composite) and visibility as \(\text{R1}\) and \(\text{R2}\). The association ends shall also have qualifiers according to the qualifiers of \(\text{R1}\) and \(\text{R2}\). The name and type of these shall be according to the transformations for attributes specified below. The association shall have the stereotypes \(\text{S1}\) to \(\text{Sn}\). This transformation makes it easy to formulate rules on associations; as for instance, the example rule introduced in section four can now be modelled as shown in Fig. 17. Contrast this with the model in Fig. 14 to see the difference from modelling using only the fundamental transformations.

![Architectural rules model](image)

![Architectural rules profile](image)

**Fig. 17** Example of rules on associations

T11. If \(\text{MA}\) equals “Dependency”, “Generalization” or “Realization” and the association is only navigable from \(\text{C1}\) to \(\text{C2}\):

A \(<\text{C}1\)> element shall have a relationship according to \(\text{MA}\) to \(\text{Mu2}\) number of \(<\text{C}2\)> elements with stereotypes \(\text{S1}\) to \(\text{Sn}\).

Examples of these kinds of transformation are shown in Fig. 18.
T12. If there are attributes A of C1 that starts with $ then:
   a. All parts of the definition of an attribute of a <<C1>> class must match the corresponding part of an A, where the wild card characters “@” and “%” in any part of the definition of A can be replaced with any character sequence. Parts of A not specified (as for instance default value for Sampling_Period in Fig. 19) are unconstrained.
   b. All A must be matched by one attribute in a <<C1>> class. An exception to this is if the name of A contains the wild card character “%”; in this case any number of matches (including zero) is allowed.
   c. If the name of a type of A is identical to the name of a class C in the architectural rules model then the type of a matching attribute must be a <<C>> element.

This transformation is exemplified in Fig. 19.

T13. If there are operations O of C1 that start with $ then:
a. All parts of the definition of an operation of a \(<\textit{C1}\rangle\) class must match the corresponding part of an \(\textit{O}\), where, for each part of the definition, the wild card characters “@” and “%” can be replaced with any character sequence. Parts of \(\textit{O}\) not specified (as for instance parameter directions for operations in Fig. 20) are unconstrained.

b. This requirement holds for all parts of the definition of \(\textit{O}\) defined in the UML meta-model, such as for instance opaque behaviour specified for the operation.

c. The character “%” in a parameter name means that the definition of this parameter can be repeated any number of times, including zero. In these parameter definitions “%” can be replaced with any character sequence.

d. If the name of the type of \(\textit{O}\) or a parameter of \(\textit{O}\) is identical to the name of a class \(\textit{B}\) in the architectural rules model then the type of matching operations or parameters in the \(<\textit{C1}\rangle\) class must be of a \(<\textit{B}\rangle\) Class.

e. All \(\textit{O}\) must be matched by one operation in a \(<\textit{C1}\rangle\) class. An exception to this is if the name of \(\textit{O}\) contains the wild card character “%”; in this case any number of matches (including zero) is allowed.

This transformation is exemplified in Fig. 20.

T14. If \(\textit{C1}\) has a state machine then a \(<\textit{C1}\rangle\) class must have a state machine where there for each region in \(\textit{C1}\) shall be an identical region in the \(<\textit{C1}\rangle\) class. The wild card character “@” may be used in the transition definitions in
An approach for modelling architectural design rules in UML

C1 and shall then be matched with any text string in the corresponding transition in the state machine of a <<C1>> class. It is allowed to have additional regions in the state machine of a <<C1>> class.

This transformation is exemplified in Fig. 21. In this example a <<Sensor>> class is constrained to have a top region exactly matching the state machine for Sensor in the architectural rules model, which in effect forces Sensor classes to call the operation Sample() periodically with the period specified by the attribute Sampling_Period. A <<Sensor>> class may have additional behaviour specified in parallel regions to the one specified in Sensor.

Architectural rules model

Sensor

stm Sensor

Sampling

Transformation

Architectural rules profile

<<Stereotype>>

Sensor

(A <<Sensor>> Class shall have a state machine with a top level region with a state machine that is a copy of the state machine of Sensor)

Fig. 21 Example on rules on state machines

These additional UML specific set of transformations make it easy to specify constraints on for instance how different kinds of classes may be associated. To illustrate, let us revisit the previously used example in section 2.2:

A sensor may only have associations to In_Port_Ifc and Data_Items. These associations shall only be navigable from the sensor.

This rule may now be modelled according to Fig. 22\textsuperscript{5}, which is very close to the simple (but only indicative) model in Fig. 5, and significantly less complex than the model in Fig. 14, where only the general transformations were used.

\textsuperscript{5} Actually, in our example introduced in section 3, Sensor would instead inherit the rule to be allowed to have associations to <<Data_Item>> from Data_Item_Observer shown in Fig. 7
These additional transformations also make it simple to specify other common constraints such as on package structure and on interfaces and the behaviour of classes. This is further illustrated in section seven.

6 Automating the Approach

In this section we present the tool for automating enforcement of architectural design rules that was developed in response to our second research objective (which was to verify that the approach was stringent enough to be automated). To provide a proof-of-concept of the feasibility of automating the approach, a tool was built making it possible to automatically check that a system conforms to rules modelled according to the approach.

There were several options when considering tool support for the approach:

- The rules could be enforced as a separate test, reporting violations.
- The rules could be continuously enforced during modelling giving the possibility of guiding the developer during development and if desired preventing the modeller from breaking the rules.
- In both cases the modelled rules could either first be transformed into OCL constraints in a UML profile, which would then be enforced, or they could be directly enforced on the system model.

An important thing to consider was how to make it as easy as possible for an organisation to adopt the new method and tool. For an organisation that is already using modelling tools it would be a big advantage if they did not have to change their modelling tools. New tools would incur cost in purchase, training and transferring models to the new tools. In our case the organisation was using the Rhapsody modelling tool. This was also the tool that had been used in building the system for which we
intended to remodel the architectural design rules. Therefore we needed a tool that could take Rhapsody models both for the architectural design rules and for the system models. Considering this we built the tool as a stand-alone checker to the Rhapsody tool validating the system model directly against the modelled rules for the following reasons:

1. To make a plug-in to the modelling tool that continuously checks the model would be harder than to make a stand-alone checker, would be harder to move to another tool, and would risk increasing the response time when modelling.

2. Although open-source stand alone tools for OCL checking are available, making an OCL generator and integrating an OCL checker in Rhapsody was considered at least as hard as our current approach and in addition would increase the risk since we would be relying on another tool.

The tool is built in C++ and is currently limited to reading Rhapsody [Telelogic] models, both for architectural rules and for the system model. The tool is designed so that there are no dependencies to the model reader component from any other parts in the tool. This makes it a relatively small task to adapt the tool to another modelling tool.

The total effort to build the tool was approximately 200 hours; the estimated effort to build another model reader is about 40 hours. A screenshot of the tool is shown in Fig. 23 where the output from the validator is shown in the text window in the bottom. The text refers to the violations in the Water_Level class in respect to the architectural rules in Fig. 7.
In this section we present the findings of a case study performed in response to our third research objective (which was to demonstrate that the approach was applicable to a real development project).

To demonstrate the applicability of our approach to a real problem we modelled the architectural design rules from an already developed system according to our set of transformations\(^6\). The mapping between the original architectural rules and the models was documented in a text table. A part of this table is shown in Table . The rules could

\(^{6}\) Note that in a real case the architectural rules model would be modelled as a natural part of the architectural design and not as a separate activity. Normally there would not even be any textual expression of the rules, only the architectural rules model.
be classified into three categories: structural, behavioural and judgmental. Structural rules specified structural constraints such as rule 4.1, 4.2, 6.7 and 8.1 in the table. Behavioural rules specified constraints on behaviour such as rule 9.16 in the table. Judgemental rules were rules where the developer had to exercise judgement to follow the rule; 3.2 is an example of such a rule in the table. There were 66 rules in total; eight of these could not be modeled. These were all judgmental and therefore inherently impossible to formalize. The rules typically consisted of one or two sentences, where the sample rules in Table 1 are representative.

Table 1. A subset of the full table which had mappings between all the original architectural design rules and resulting modelling constructs

<table>
<thead>
<tr>
<th>Id</th>
<th>Original rule (Quotation)</th>
<th>Modelreference</th>
<th>Used Transformations</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2</td>
<td>“Functionality specific to a PAPI requirement shall be kept in this layer unless it is reusable for another PAPI or applicable to DVB standard. In this case it shall be placed in CMP or CMD.”</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>4.1</td>
<td>“All coupling between arcComponents shall be loose in the sense not statically linked”</td>
<td>Handled by only allowing associations from a component to an Interface that is realized by an arcComponent.</td>
<td>T9, T10, T11</td>
</tr>
<tr>
<td>4.2</td>
<td>“All associations between arcComponents shall be navigable from the client to the server (user to the resource)”</td>
<td>User/Resource association from mComponent to mCompIfc</td>
<td>T10</td>
</tr>
<tr>
<td>6.7</td>
<td>“In the case of a component locked to a specific arcComponentUser, it is the responsibility of the locker to allow only one thread at a time to access the component.”</td>
<td>This is ensured by the implementation of the enforced implementation of the operations of the mLockableComponent.</td>
<td>T13</td>
</tr>
<tr>
<td>8.1</td>
<td>“All locked components shall inherit the same base class, arcLocked.”</td>
<td>Generalization from mLockableComponent to marcLockableComponent, there is only one instance of marcLockableComponent allowed, in an Architecture_Pkg and finally there is only one marcLockableComponent class in the Systemmodel.</td>
<td>T8, T11, T13</td>
</tr>
<tr>
<td>9.16</td>
<td>“Transmission events and exceptions initiated by a Write() shall be reported back to the arcPortUser via the TxDone() call.”</td>
<td>A Write operation is forced to always end with a call to TxDone</td>
<td>T13</td>
</tr>
</tbody>
</table>

The average size of the rules was 17 words with a maximum of 38 words and a minimum of four words. Table 2 shows for each transformation the number of rules it was used to model (Usage frequency) and the percentage of architectural review remarks in the review protocols that related to these rules (Violations).
transformations T1 and T9 are marked as not applicable since T1 is always used and T9 is always used in conjunction with T10 and T11. In total there were 1563 remarks in 120 architectural review protocols. The table shows that the most commonly used rules were T8, T10, T11 which specify structural rules on package containment, associations, relations and generalizations, and T13 which specify rules on the operations (i.e. the interface) of classes. Not surprisingly, these rules are also subject to most violations.

Table 2. Usage frequency and violation percentage for each transformation

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T4</th>
<th>T5</th>
<th>T6</th>
<th>T7</th>
<th>T8</th>
<th>T9</th>
<th>T10</th>
<th>T11</th>
<th>T12</th>
<th>T13</th>
<th>T14</th>
</tr>
</thead>
<tbody>
<tr>
<td>Usage frequency</td>
<td>N.A.</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>10</td>
<td>N.A.</td>
<td>13</td>
<td>22</td>
<td>6</td>
<td>17</td>
<td>5</td>
</tr>
<tr>
<td>Violations (%)</td>
<td>N.A.</td>
<td>3.37</td>
<td>4.46</td>
<td>2.92</td>
<td>0.00</td>
<td>3.19</td>
<td>1.55</td>
<td>15.38</td>
<td>N.A.</td>
<td>20.73</td>
<td>34.84</td>
<td>8.55</td>
<td>25.29</td>
<td>9.57</td>
</tr>
</tbody>
</table>

Both the architectural rules model and the architectural parts of the system model were captured in the Rhapsody modelling tool (version 7.2). Fig. 24 - Fig. 26 show parts of these models. Fig. 25 shows the subsystems modelled as packages in the system model. This level of the system model is owned by the architects. The stereotypes of these packages are defined in the architectural rules model partly shown in Fig. 24. In this model you can see, for instance, the architectural rules that a <<Subsystem>> package (that is, a package with the stereotype <<Subsystem>>) shall contain a number of <<Component_Pkg>> packages and one <<mRegistry>> class. You can also see that a <<Component_Pkg>> shall contain exactly one <<mComponent>> class that must inherit a <<marcComponent>> class (defined by the architects in the architecture package in the system model.). In Fig. 26 an example of a small component in the system model is shown, following the architectural rules defined in the architectural rules model.
Fig. 24 Part of the architectural rules model

Fig. 25 Top level of system model.
8 Discussion and Conclusions

Architectural design rules are an important part of the architecture and there are no adequate solutions in the current body of literature on how to model them. The inability to formalize the architectural design rules leads to a need for error-prone and time-consuming manual tasks to enforce them. The approach developed in this study addresses this problem by providing a technique for modelling architectural design rules in a way that is formal enough to allow automation. An important property of the approach is that the architectural design rules are modelled using UML at a high abstraction level, without requiring detailed knowledge of the UML meta-model. That the rules are modelled at an abstraction level close to that of the rule itself is required for the models to be easily understandable for architects and developers, an issue of paramount importance for the usability of the approach. The use of UML reduces the required investment in tools and training since architects and developers benefit from previous knowledge in UML and are able to use their current UML tools for modelling; to provide automation only requires an additional tool that checks the system model against the architectural model according to our defined transformations. Our effort for
building such a tool for the Rhapsody modelling tool using its COM API was approximately 200 man-hours, so this should be a relatively small task.

In applying our approach to modelling the architectural design rules of an industrial strength system, we found that of the original 66 rules only eight could not be modelled. This means that we would have relieved the architects of a large part of their enforcement effort; only 12% of the rules would have been left for manual enforcement. The rules that could not be modelled were all rules where the developer was supposed to exercise judgement, which made them inherently impossible to formalize. A typical example of such a rule was:

“Functionality specific to a PAPI requirement shall be kept in this layer unless it is reusable for another PAPI or applicable to the DVB standard. In this case it shall be placed in CMP or CMD.”

These are rules that need a lot of interaction between the developers and the architects in order to develop a common understanding of what the rules really mean. It is very important to get this right at the same time as it is impossible to finalize and formalize them at an early stage in the project. This is where the focus of the architects should be and our approach gives the architects the time to do that. Other benefits are that modelling eliminates ambiguities and redundancy in the rules which should make them easier to understand and give less room for erroneous interpretations.

Although the approach has only been tested on one system, two factors suggest that the results should, to a large extent, be transferable to other systems and organisations in the embedded software domain:

1. The defined transformations are based on raising the general modelling constructs of UML to the meta-model level, not on the specific needs of the system used for the test.
2. It is a real-world embedded system of significant size with functionality quite common in this domain.

There is a need for further research to study the implications when adopting the approach in other application domains. Factors to investigate include the ease with
which architects, developers and other stakeholders can learn the approach and accommodate their working practices to it.

Appendixes

Appendix A. Formal definition of transformations

In this appendix the transformations described in section 5 is formally defined as a MOFScript transformation. The transformation takes two input models and produces one uml profile and one OCL definition file. The first input model (bound to mdl in the script) shall be a model containing both the architectural rules model and the system model where the architectural rules model shall be in a package named “Arch_Rules”. The second input model (bound to Langmdl in the script) shall be the UML meta model. Both of these models shall be in the form of eclipse uml2 models (http://www.eclipse.org/modeling/mdt/?project=uml2). The output profile is stored in a file named “Arch_Rules_Profile.uml” and the OCL constraints are stored in a file named "Arch_Rules_Profile.ocl". The script has been tested with MOFScript version 1.3.8 (http://www.eclipse.org/gmt/mofscript/) and the resulting profile and OCL constraints have been tested with the TOPCASED modelling tool version 3.3.0 (http://www.topcased.org/) using the TOPCASED OCL Checker. The input model containing the architectural rules and the system model was designed in the TOPCASED modelling tool and as input UML metamodel the “model/uml.merged.uml” file found in “org.eclipse.uml2.uml.source_3.0.1.v200908281330.jar” java archive file in the plugins directory of eclipse was used.

The constraints are not defined in the context of the stereotypes, which probably would have been the most straight forward way of definition; instead they are defined in the context of the metaclass extended by the stereotype. This means that a test has to be added to the constraint in order to only execute the actual constraint if the instance of the metaclass has the extending stereotype. The reason for this strategy is simply that we have not found any way in the TOPCASED toolkit of defining OCL helper functions so that they are available from stereotypes defined within stereotypes and since some of the constraints use recursion this is required. However, although this strategy may seem a bit awkward it does not make the defined constraints any less suitable for their purpose in this context; to formally define the meaning of the
Another strategy that we have chosen is to define the constraints as relations between the system model and the architectural rules model as far as possible. This has been used for transformations T8 to T14 which means that the actual transformations for these are trivial, they only invoke an OCL helper function with the name of the stereotype (i.e. inv xxx: ValidateT10(‘Sensor’) as can be seen in the Main transformation). The major benefit of this approach is that the transformation can be defined purely in OCL instead of partly in MOFscript and partly in OCL. We believe this makes them easier to understand. This also means that the resulting OCL file is smaller and that a significant part of it is constant. This constant part is generated by the MOFScript rule “GenerateOCLHelpers()” defined in the end of the MOFscript script. This transformation only contains a large part of constant escaped output in which the definitions of transformations T8 to T14 can be found as OCL helper functions named ValidateTx(). The remaining transformations, T1 to T7 are handled in the following way:

- T1 is handled by the transformation Add_Stereotype(…) which adds a stereotype to the profile.
- T2 is mainly handled by an “if” clause in the Main transformation that prevents transformation of classes stereotyped <<metaclass>> but there are also some implications for how the other transformations are defined.
- T3 is handled by the transformation “Do_Transform_T3(…)”.
- T4 and T5 are handled by the transformation “Do_Transform T4_5(…)”.
- T6 is handled by the transformation “Do_Transformation_T6(…)”.
- T7 is mainly handled by the transformation “Create_Generalizations(…)” but there are also some implications for how the other transformations are defined.

The OCL helper functions WCMatch(…) and RegMatch(…) are only included in order to make the resulting OCL file executable by the TOPCASED OCL Checker. They implement wildcard matching and regular expression matching for strings and should normally be provided in some other way, typically using external library functions. While the implementation of WCMatch provided here actually works as intended the implementation of RegMatch only allows simple wildcard matching (using WCMatch).
An approach for modelling architectural design rules in UML

texttransformation ARV_Transformation {
  in mdl:"http://www.eclipse.org/uml2/3.0.0/UML",
  in Langmdl:"http://www.eclipse.org/uml2/3.0.0/UML",
  out ARP:"http://www.eclipse.org/uml2/3.0.0/UML"{ARP.Profile}){
    main () {
      var Prof:ARP.Profile = new ARP.Profile(name="Arch_Rules_Profile")
      var MetaModel:ARP.PackageImport = new ARP.PackageImport(importedPackage=Langmdl)
      var Rules_Package:mdl.Package = mdl.objectsOfType(mdl.Package)->select(P:mdl.Package|P.name = "Arch_Rules").first()
      file("Arch_Rules_Profile.ocl")
      GenerateOCLHelpers()
      Prof.metamodelReference.add(MetaModel)
      Rules_Package.getAllClasses()->forEach(C:mdl.Class) {
        var M:String = getClassStereotype(C)
        var S:ARP.Stereotype
        if (M.toLower() != "metaclass") {
          S=Prof.Add_Stereotype(C.name,M)
          Do_Transform_T3(C,M)
          Do_Transform_T4_5(C,M,S)
          Do_Transform_T6(C,M,S)
          if (M=="Package") {
            'context Package\n            'inv T8_'C.name": ValidateT8("C.name")\n          } else if (M=="Class") {
            'context Class\n            'inv T10_'C.name": ValidateT10("C.name")\n            'inv T11_Dep_'C.name": ValidateT11_Dep("C.name")\n            'inv T11_Realization_'C.name": ValidateT11_Realization("C.name")\n            'inv T11_Generalization_'C.name": ValidateT11_Generalization("C.name")\n            'inv T12_'C.name": ValidateT12("C.name")\n            'inv T13_'C.name": ValidateT13("C.name")\n            'inv T14_'C.name": ValidateT14("C.name")\n          }
        }
      }
      Prof.Create_Generalizations(Rules_Package)
      Prof.store("Arch_Rules_Profile.uml")
    }
  }
}
mdl.Package::getAllClasses():List {
  var L:List = self.packagedElement->select(C:mdl.Class)
  self.packagedElement->forEach(P:mdl.Package) {
    L.addAll (P.getAllClasses())
  }
  return L
}
(getClassStereotype(C:mdl.Class):String {
  if (C.getAppliedStereotypes().isEmpty()) {
  ...
return "Class"
} else {
    return C.getAppliedStereotypes().first().name
}
}

ARP.Profile::Add_Stereotype(StereotypeName:String,MetaclassName:String)::ARP.Stereotype
{
    var MetaClass:Langmdl.Class = getMetaClass(MetaclassName)
    var S:ARP.Stereotype = new ARP.Stereotype(name=StereotypeName)
    var Ext:ARP.Extension = new ARP.Extension()
    var BaseEnd:ARP.Property = new ARP.Property()
    var ExtEnd:ARP.ExtensionEnd = new ARP.ExtensionEnd()

    self.Import_MetaClass(MetaClass)
    self.ownedStereotype.add(S)
    self.packagedElement.addOrg(Ext)

    Ext.name=MetaClass.name+"_"+S.name
    ExtEnd.name="extension_"+StereotypeName
    ExtEnd.isComposite =true
    ExtEnd.type =S
    BaseEnd.name="base_"+MetaClass.name
    BaseEnd.type = MetaClass
    S.ownedAttribute.addOrg(BaseEnd)
    Ext.ownedEnd.addOrg(ExtEnd)
    Ext.memberEnd.addOrg(BaseEnd)
    return S
}

Do_Transform_T3(C:mdl.Class,M:String) {
    var OtherEnds : List
    var Roles : Hashtable
    var T12_Apply: Boolean = !(C.ownedAttribute->select(P:mdl.Property|P.name.startsWith("$"))*(P).isEmpty())
    var T13_Apply: Boolean = !(C.ownedOperation->select(P:mdl.Property|P.name.startsWith("$"))*(P).isEmpty())
    C.getAllEnds()->forEach(E : mdl.Property) {
        if (not E.getAppliedStereotypes().isEmpty()) {
            var RoleName:String = E.getAppliedStereotypes().first().name
            if (!((M="Class" and (T12_Apply and RoleName = "ownedAttribute") or (T13_Apply and RoleName = "ownedOperation")))) {
                Roles = E.AddEnd(RoleName,Roles)
            }
        }
    }
    Roles.keys()->forEach(R:String) {
        var L :List = Roles.get(R)
        var MetaClass:String = getMeta_Class(L.first().type)
        var SetNum : Integer =0
        var Prefix:String="" +
        var SubtractedSets:String = ""
L = T2Sort(L)
'context ' M'\n' \n'inv T3_' C.name '_' R' : \n'
"if hasStereotype("C.name") then\n"

L->forEach(E:mdl.Property) { 
    var SelectCondition:String
    SetNum=SetNum+1
    if (E.type.isMetaClass()) { 
        SelectCondition = "oclIsKindOf(" + MetaClass + ")"
    } else { 
        SelectCondition = "isOfStereotype(" + E.type.name + ")"
    }
    if (SetNum = 2) { 
        Prefix = Prefix + "and "
    }
    if (SetNum > 1) { 
        SubtractedSets = SubtractedSets + "-Set" + (SetNum-1)
    }
    Prefix'let Set'SetNum':Set('MetaClass')='R'
        -select('SelectCondition')'SubtractedSets' in \n'

    'E.lower' <= Set'SetNum'->size() and Set'SetNum'->size() <= 'E.upper'\n'
'else true endif\n"
}
//------------------------------------------------------------------
Do_Transform_T4_5(C:mdl.Class,M:String,S:ARP.Stereotype){
    var MetaClass:Langmdl.Class = getMetaClass(M)
    if (MetaClass!=null) {
        C.ownedAttribute->forEach(A:mdl.Property |not A.name.startsWith("\$")) { 
            if (AttributeIsInMetaClass(A,MetaClass)) { 
                "context " M "\n'
                "inv T4."C.name": isOfStereotype("C.name") implies 
                "A.name".asString().MatchReg("A.defaultValue.value")\n'
            } else { 
                S.ownedAttribute.add(A)
            }
        }
    }
//------------------------------------------------------------------
Do_Transform_T6(C:mdl.Class,M:String,S:ARP.Stereotype) {
    var Specs:List = C.ownedRule->select(tmp:mdl.Constraint)
C.ownedRule-
>forEach(Con:mdl.Constraint|Con.specification.oclIsTypeOf(mdl.OpaqueExpression)) {
    var Sp:mdl.OpaqueExpression = Con.specification
    var OCLConstraint:ARP.Constraint = new ARP.Constraint(name=Con.name)
    Sp.language->forEach(b:String) {Spec.language.add(b)}
    Sp.body->forEach(b:String) {Spec.body.add(b)}
    OCLConstraint.specification = Spec
    S.ownedRule.add(OCLConstraint)
}

ARP.Profile::Create_Generalizations(Rules_Package:mdl.Package) {
    Rules_Package.getAllClasses()>
        forEach(Sub:mdl.Class|not Sub.superClass.isEmpty()){var SubStereotype:ARP.Stereotype = self.ownedStereotype->
        select(S:ARP.Stereotype|S.name==Sub.name).first()}
    Sub.superClass->forEach(Super:mdl.Class) {
        var SuperStereotype:ARP.Stereotype = self.ownedStereotype->
        select(S:ARP.Stereotype|S.name==Super.name).first()}
    SubStereotype.superClass.add(SuperStereotype)

getMetaClass(Cname:String):Langmdl.Class {
    var Mcl:List = Langmdl.objectsOfType(Langmdl.Class)->
        select(c: Langmdl.Class | c.name.equals(Cname))
    if (Mcl.isEmpty()) {
        return
    } else {
        return Mcl.first()
    }
}

AttributeIsInMetaClass(A:mdl.Property, M:Langmdl.Class) : Boolean {
    var Alist : List = M.getAllAttributes() -> select(MA:Langmdl.Property | MA.name=A.name)
    return not Alist.isEmpty()
}

Langmdl.Class::getAllAttributes():List {
    var L:List
    self.ownedAttribute->forEach(a:Langmdl.Property) {
        L.add(a)
    }
    self.superClass->forEach(c:Langmdl.Class) {
        L.addAll (c.getAllAttributes())
    }
    return L
}

ARP.Profile::Import_MetaClass(Mc:ARP.Class) {
    var found: boolean = false
self.metaClassReference->forEach(R:ARP.ElementImport){
  if (R.importedElement==Mc) {
    found = true
  }
}
if (not found) {
  var MetaClassRef:ARP.ElementImport = new ARP.ElementImport(importedElement=Mc)
  self.metaClassReference.add(MetaClassRef)
}

getMeta_Class(C:mdl.Class):String {
  var M :String = getClassStereotype(C)
  if (M.toLower()="metaclass") {
    return C.name
  } else {
    return M
  }
}
mdl.Class::isMetaClass():Boolean {
  return self.hasStereotype("metaclass")
}
MkList(O:Object):List {
  var L:List
  L.add(O)
  return L
}
mdl.Property::AddEnd(Key:Object,H:Hashtable):Hashtable {
  var tmp : Hashtable = H
  if (H.get(Key)==null) {
    tmp.put(Key,MkList(self))
  } else {
    tmp.get(Key).add(self)
  }
  return tmp
}
mdl.Class::getTypeLevel():Integer {
  var Max: Integer = 0
  if (self.isMetaClass()) {
    return 0
  } else {
    self.superClass->forEach(C:mdl.Class) {
      var L:Integer = C.getTypeLevel()
      if (L > Max ) {
        Max = L
      }
    }
  }
  return Max + 1
T2Sort(L:List):List {
    var TypeLevelTable:Hashtable
    var N:List
    var MinLevel: Integer = 10000
    var MaxLevel: Integer = 0
    var Level: Integer
    L->forEach(E:mdl.Property) {
        Level = E.type.getTypeLevel()
        TypeLevelTable = E.AddEnd(Level, TypeLevelTable)
        if (Level > MaxLevel) {
            MaxLevel = Level
        }
        if (Level < MinLevel) {
            MinLevel = Level
        }
    }
    Level = MaxLevel
    while (Level >= MinLevel) {
        if (TypeLevelTable.get(Level) != null) {
            N.addAll(TypeLevelTable.get(Level))
        }
        Level = Level - 1
    }
    return N
}

mdl.Class::getAllEnds():List {
    var Lst:List
    self.getAssociations() -> forEach(A:mdl.Association) {
        A.memberEnd -> forEach(E:mdl.Property | E.type<> self) {
            Lst.add(E)
        }
    }
    self.superClass -> forEach(C:mdl.Class) {
        Lst.addAll(C.getAllEnds())
    }
    return Lst
}

// GenerateOCLHelpers()!

MainModel : http://www.eclipse.org/uml2/3.0.0/UML

context uml::String
def: asString():String = self

class context uml::Boolean
def: asString():String =
    if self then 'true' else 'false' endif
context uml::Integer
def: asString():String=
if self < 0 then '-'.concat(self.abs().asString())
else
let i :Integer = self.mod (10) in
(self.div(10)).asString().concat(
  if i=0 then '0'
  else if i=1 then '1'
  else if i=2 then '2'
  else if i=3 then '3'
  else if i=4 then '4'
  else if i=5 then '5'
  else if i=6 then '6'
  else if i=7 then '7'
  else if i=8 then '8'
  else '9'
endif)
endif
endif
-------------------------------------------------------------------------------------
context uml::String
def isWC: isWC(i:Integer):Boolean =
let c:String= self.substring(i,i) in c='@' or c='%' or c='*
---------------------------------------------
context uml::String
def FindS: FindS(Pattern:String,Pos:Integer): Integer =
let Psize:Integer = Pattern.size() in
let Ssize:Integer = self.size() in
if Pattern.isWC(1) then
  if Psize=1 then Pos
  else
    Sequen
case{Pos..Ssize}-iterate(i:Integer;EndPos:Integer=0|
      if EndPos<>0 then EndPos --done
      else
        if Psizen= Ssize-1 then -1 --failed
        else
          if self.substring(i,i+Psize-2) = Pattern.substring(2,Psize) then i+Psize-1
          else
            if i=Ssize then -1 --failed
            else 0 --keep on looking
          endif
        endif
      endif
    }
  endif
endif
context uml::String
def WCMatch: WCMatch(Pattern:String): Boolean =
  let End:Integer = self.size() in
  let WcPositions:OrderedSet(Integer) = Sequence{1..Pattern.size()}->
    select(i:Integer|Pattern.isWC(i)) in
  let Patterns:OrderedSet(String) = Sequence{0..WcPositions->size()}->
    iterate(i:Integer;acc:OrderedSet(String)=OrderedSet{}|
      if i=0 then
        if WcPositions->size()==0 then
          acc->append(Pattern)
        else
          if WcPositions->at(1) > 1 then
            acc->append(Pattern.substring(1,WcPositions->at(1)-1))
          else acc
          endif
        endif
      else
        if i=WcPositions->size() then
          acc->append(Pattern.substring(WcPositions->at(i),Pattern.size()))
        else
          acc->append(Pattern.substring(WcPositions->at(i),WcPositions->at(i+1)-1))
        endif
      endif
    )in
  let Res:Integer =Sequence{1..Patterns->size()}->iterate(i:Integer;Pos:Integer=1|
    if Pos= -1 then -1
  else self.FindS(Patterns->at(i),Pos)
  endif
) in
  if WcPositions->isEmpty() then
    self = Pattern
  else
    if Res< 1 then false
    else true
  endif
enddef
context uml::String
def MatchReg: MatchReg(RegExp:String):Boolean =
  --In a full implementation this should be replace with a true regexp matching function
  self.WCMatch(RegExp)
enddef
context NamedElement
def IsRuleElement: IsRuleElement():Boolean =
if oclIsKindOf(Class) then
  oclAsType(Class).allOwningPackages() -> exists(name='Arch_Rules')
else false
endif

context Element
def getStereotype(): String = self.getAppliedStereotypes() -> any(true).name

context Element
def hasStereotype(S: String): Boolean =
  self.getAppliedStereotypes() -> exists(name=S)

context Element
def hasSameStereotypes(S: String, E: Element): Boolean =
  self.getAppliedStereotypes() -> forAll(s:Stereotype|E.hasStereotype(s.name)) and
  E.getAppliedStereotypes() -> forAll(s:Stereotype|self.hasStereotype(s.name) or s.name=S)

context Element
def isOfMetaClass(S: String): Boolean =
  if self.eClass().name = S then true
  else self.eClass().eAllSuperTypes -> exists(name=S)
endif

context Class
def allSuperClasses(): Set(Class) =
  if superClass -> isEmpty() then Set{}
  else superClass -> union(superClass.allSuperClasses() -> asSet())
endif

context Element
def isOfStereotype(S: String): Boolean =
  let Applied: Set(Class) = getAppliedStereotypes() in
  Applied -> union(Applied.allSuperClasses() -> asSet()) -> exists(name=S)

context NamedElement
def MatchRuleClass(R: NamedElement): Boolean =
  if R.hasStereotype('metaclass') then
    self.isOfMetaClass(R.name)
  else
    self.isOfStereotype(R.name) and
    if R.getStereotype().oclIsUndefined() then
      self.isOfMetaClass('Class')
    else
      self.isOfMetaClass(R.getStereotype())
    endif
  endif
endif

context Class
def getAllEnds(SName: String): Set(Property) =
  getAssociations() -> select(hasStereotype(SName)).memberEnd -> select(type<>self) ->
  union(self.superClass.getAllEnds(SName)) -> asSet()
context Class
def getAllEnds: getAllEnds():Set(Property) =
 getAssociations().memberEnd->select(type<>self)->union(self.superClass.getAllEnds())->asSet()
-------------------------------------------------------------------------------------
context Dependency
def isDepOk: isDepOk(R:Property) : Boolean =
supplier->any(true).MatchRuleClass(R.type) and hasSameStereotypes('Dependency',R.association)
-------------------------------------------------------------------------------------
context Class
def ValidateT11_Dep: ValidateT11_Dep(Sname :String):Boolean =
if self.hasStereotype(Sname) then
  let R:Class = Class.allInstances()->any(name=Sname and IsRuleElement()) in
  let Rules:Set(Property) = R.getAllEnds('Dependency')->select(isNavigable()) in
  let AllDeps: Set(Dependency) = self.clientDependency->select(oclIsTypeOf(Dependency)) in
  Rules->forall(r:Property|r.includesCardinality(AllDeps->select(isDepOk(r))->size()))
and
  AllDeps->forall(d:Dependency|Rules->exists(r:Property|d.isDepOk(r)))
else true
endif
-------------------------------------------------------------------------------------
context Dependency
def isRelOk: isRelOk(R:Property) : Boolean =
supplier->any(true).MatchRuleClass(R.type)
-------------------------------------------------------------------------------------
context Class
def ValidateT11_Realization: ValidateT11_Realization(Sname :String):Boolean =
if self.hasStereotype(Sname) then
  let R:Class = Class.allInstances()->any(name=Sname and IsRuleElement()) in
  let Rules:Set(Property) = R.getAllEnds('Realization')->select(isNavigable()) in
  let AllReal: Set(Dependency) = self.clientDependency->select(oclIsKindOf(Realization)) in
  Rules->forall(r:Property|r.includesCardinality(AllReal->select(isRelOk(r))->size()))
and
  AllReal->forall(d:Dependency|Rules->exists(r:Property|d.isRelOk(r)))
else true
endif
-------------------------------------------------------------------------------------
context Class
def isGenOk: isGenOk(R:Property) : Boolean =
MatchRuleClass(R.type.oclAsType(Class))
-------------------------------------------------------------------------------------
context Class
def ValidateT11_Generalization: ValidateT11_Generalization(Sname :String):Boolean =
if self.hasStereotype(Sname) then
  let R:Class = Class.allInstances()->any(name=Sname and IsRuleElement()) in
  let Rules:Set(Property) = R.getAllEnds('Generalization')->select(isNavigable()) in
  let AllGen: Set(Class) = self.superClass in
An approach for modelling architectural design rules in uml

Rules->forAll(r:Property|r.includesCardinality(AllGen->select(isGenOk(r))->size()))
and
AllGen->forAll(c:Class|Rules->exists(r:Property|c.isGenOk(r)))
else true
endif

context Property
def getOppEnd: getOppEnd():Property =
association.memberEnd->any(p|p<>self)

context Property
def QualifiersMatch: QualifiersMatch(R:Property):Boolean =
let AllQ: OrderedSet(Property) = self.qualifier in
let Rules: OrderedSet(Property) = R.qualifier in
AllQ->size()=Rules->size() and
Sequence{1..Rules->size()}->forAll(i:Integer|
    let r:Property = Rules->at(i) in
    let a:Property = AllQ->at(i) in
    a.name.WCMatch(r.name) and
    if r.type->isEmpty() then true
else if r.type.IsRuleElement() then a.type.isOfStereotype(r.type.name)
else a.type.name.WCMatch(r.type.name)
endif
diff
)

context Property
def isAssOk: isAssOk(R:Property) : Boolean =
let Ao:Property = self.getOppEnd() in
let Ro:Property = R.getOppEnd() in
type.MatchRuleClass(R.type) and
self.isNavigable() = R.isNavigable() and
Ao.isNavigable() = Ro.isNavigable() and
self.aggregation = R.aggregation and
Ao.aggregation = Ro.aggregation and
self.visibility = R.visibility and
Ao.visibility = Ro.visibility and
self.QualifiersMatch(R) and
Ao.QualifiersMatch(Ro)

context Class

def ValidateT10: ValidateT10(Sname :String):Boolean =
if self.hasStereotype(Sname) then
    let R:Class = Class.allInstances()->any(name=Sname and IsRuleElement()) in
    let Rules:Set(Property) = R.getAllEnds('Association') in
    let AllAss: Set(Property) = self.getAssociations().memberEnd->
        select(type<>self)->asSet() in

    Rules->forAll(r:Property|r.includesCardinality(AllAss->select(isAssOk(r))->size()))
and
AllAss->forAll(p:Property|Rules->exists(r:Property|p.isAssOk(r)))
else true
endif

context PackageableElement
def PkgElemOk: PkgElemOk(Rule:Property):Boolean =
self.MatchRuleClass(Rule.type) and self.visibility = Rule.visibility

context Package
def ValidateT8: ValidateT8(Sname :String):Boolean =
if self.hasStereotype(Sname) then
  let R:Class = Class.allInstances()->any(name=Sname and hasStereotype('Package') and
    IsRuleElement()) in
  let Rules:Set(Property) = R.getAllEnds()->select(isComposite) in
  let AllContent: Set(PackageableElement) = self.packagedElement in
  Rules->forAll(r:Property|r.includesCardinality(AllContent->
    select(PkgElemOk(r))->size())) and
  AllContent->forAll(p:PackageableElement|Rules->exists(r:Property|p.PkgElemOk(r)))
else true
endif

class VisibilityKind
def VisibilityasString: asString() : String =
if self=VisibilityKind::public then '+'
else if self=VisibilityKind::protected then '#'
else if self=VisibilityKind::private then '-'
else '--'
endif

context NamedElement
def getVisibility: getVisibility():String =
if visibility->isEmpty() then '@'
else visibility.asString()
endif

class MultiplicityElement
def getMult: getMult():String =
if (lowerValue->isEmpty() or upperValue->isEmpty()) then '@'
else lowerValue.stringValue().concat('..').concat(upperValue.stringValue())
endif

context NamedElement
def Opt: Opt():Boolean =
Sequence{1..name.size()}->exists{i:Integer|name.substring(i,i)='%'}

context Property
def getDef: getDef():String =
if (default->isEmpty()) or default.size()==0 then '@'
else true
context Property
  def AttIsOk: AttIsOk(R:Property):Boolean =
  name.WCMatch(R.name.substring(2,R.name.size()))
  and getVisibility().WCMatch(R.getVisibility())
  and getDef().WCMatch(R.getDef())
  and getMult().WCMatch(R.getMult())
  and (R.isStatic implies isStatic)
  and
  if R.type->isEmpty() then true
  else if R.type.IsRuleElement() then self.type.isOfStereotype(R.type.name)
  else self.type.name.WCMatch(R.type.name)
  endif
  endif

context Class
  def ValidateT12: ValidateT12(Sname :String):Boolean =
  if self.hasStereotype(Sname) then
    let R:Class = Class.allInstances()>
    let Rules:Set(Property) = R.getAllAttributes()>
    let AllAtt: Set(Property) = self.ownedAttribute in
    if Rules->notEmpty() then
      Rules->forall(r:Property|not r.Opt()) implies AllAtt->select(AttIsOk(r)->size()==1)
    and
    AllAtt->forall(p:Property|Rules->exists(r:Property|p.AttIsOk(r)))
    else true
    endif
  else true
  endif

context Parameter
  def getDef: getDef():String =
  if (default->isEmpty()) or default.size()==0 then
    '@'
  else default
  endif

context Parameter
  def getDir: getDir():String =
  if (direction=ParameterDirectionKind::inout) then 'inout'
  else if (direction=ParameterDirectionKind::out) then 'out'
  else if (direction=ParameterDirectionKind::return) then 'return'
  else 'in'
  endif
  endif
  endif

context Parameter
def ParIsOk: ParIsOk(R:Parameter):Boolean =
name.WCMatch(R.name)
and getDef().WCMatch(R.getDef())
and getDir().WCMatch(R.getDir())
and if R.type->isEmpty() then true
else if R.type.IsRuleElement() then type.isOfStereotype(R.type.name)
else type.name.WCMatch(R.type.name)
endif
endif

context Operation
def PLMatch: PLMatch(Pt: OrderedSet(Parameter),--Architectural rules model parameter
Pl: OrderedSet(Parameter), -- System model parameters
p:Integer, --current rules parameter to match
f:Integer, --first parameter in Pl to try to match
l:Integer) --last parameter that has been matched :Boolean =
if Pt->size()=0 and Pl->size()=0 then true
else
if p>Pt->size() then -- out of rule parameters
if i=Pt->size() then -- last parameter has been matched
true
else
false
endif
else
if f>Pl->size() then -- out of model parameters, rest must be optional
Sequence{p..Pt->size()}->forAll(i:Integer|Pt->at(i).Opt())
else -- normal state
let i :Integer = Sequence{f..Pl->size().min(l+1)}->select(a:Integer|
Pl->at(a).ParIsOk(Pt->at(p)))->first() in
if i.oclIsUndefined() then -- No match
if Pt->at(p).Opt() then
PLMatch(Pt,Pl,p+1,f,l) --take next rule parameter
else
false
endif
else
let j :Integer = Sequence{i..Pl->size()}->select(a:Integer|
Pl->at(a).ParIsOk(Pt->at(p)))->last() in -- j is last matching parameter from i
if Pt->at(p).Opt() then --current rule parameter is optional
PLMatch(Pt,Pl,p+1,i,j.max(l).max(i))
else
PLMatch(Pt,Pl,p+1,i+1,i.max(j))
endif
endif
endif
endif

def OpIsOk: OpIsOk(R:Operation):Boolean =
    name.WCMatch(R.name.substring(2,R.name.size()))
    and getVisibility().WCMatch(R.getVisibility())
    and (R.isStatic implies isStatic)
    and (R.isAbstract implies isAbstract)
    and PLMatch(R.ownedParameter,self.ownedParameter,1,1,0)
    and
    let Rbody:String = R.method
    -
    any(b:OpaqueBehavior|true).oclAsType(OpaqueBehavior)._body->first() in
    if Rbody.oclIsUndefined() then true
    else
    let Obody:String = self.method
    -
    any(b:OpaqueBehavior|true).oclAsType(OpaqueBehavior)._body->first() in
    if Obody.oclIsUndefined() then false
    else Obody.WCMatch(Rbody)
    endif
    endif
----------------------------------------
context Class
def ValidateT13: ValidateT13(Sname :String):Boolean =
    if self.hasStereotype(Sname) then
    let R:Class = Class.allInstances() ->any(name=Sname and IsRuleElement()) in
    let Rules:Set(Operation) = R.getAllOperations() ->select(name.substring(1,1)='$') in
    let AllOp: Set(Operation) = self.ownedOperation in
    if Rules->notEmpty() then
    Rules->forall(r:Operation|(not r.Opt()) implies AllOp->select(OpIsOk(r))->size()==1)
    and
    AllOp->forAll(o:Operation|Rules->exists(r:Operation|o.OpIsOk(r)))
    else true
    endif
    else true
    endif
----------------------------------------
context State
def FullStateName: FullStateName():String =
    self.allNamespaces() ->iterate(n:Namespace;accName:String=self.name |
    if n.oclIsTypeOf(State) then accName.concat('::').concat(n.name)
    else accName
    endif )
----------------------------------------
context Behavior
def MatchB: MatchB(B:Behavior):Boolean =
    if B.oclIsTypeOf(OpaqueBehavior) then
    self.oclAsType(OpaqueBehavior)._body->asOrderedSet() ->first().WCMatch(
    B.oclAsType(OpaqueBehavior)._body->asOrderedSet() ->first())
    else
    true -- Only enforcement of opaque behavior supported
    endif
----------------------------------------
context Pseudostate
def EqualPS: EqualPS(P:Pseudostate):Boolean = (P.kind = kind)
context Trigger
def MatchT: MatchT(T:Trigger):Boolean =
  let e:Event = self.event in
  let r:Event = T.event in
  if e.oclIsTypeOf(AnyReceiveEvent) then
    r.oclIsTypeOf(AnyReceiveEvent)
  else if e.oclIsTypeOf(SignalEvent) then
    r.oclIsTypeOf(SignalEvent) and
    e.oclAsType(SignalEvent).signal.name.WCMatch(r.oclAsType(SignalEvent).signal.name)
  else if e.oclIsTypeOf(CallEvent) then
    r.oclIsTypeOf(CallEvent) and
    let ROpName:String = r.oclAsType(CallEvent).operation.name in
    e.oclAsType(CallEvent).operation.name.WCMatch(ROpName.substring(2,ROpName.size()))
  else if e.oclIsTypeOf(TimeEvent) then
    r.oclIsTypeOf(TimeEvent) and
    let e1:TimeEvent = e.oclAsType(TimeEvent) in
    let e2:TimeEvent = r.oclAsType(TimeEvent) in
    e1.when.stringValue().WCMatch(e2.when.stringValue()) and
    (e1.isRelative = e2.isRelative)
  else if e.oclIsTypeOf(ChangeEvent) then
    r.oclIsTypeOf(ChangeEvent) and
    let s1:String = e.oclAsType(ChangeEvent).changeExpression.stringValue() in
    let s2:String = r.oclAsType(ChangeEvent).changeExpression.stringValue() in
    s1.WCMatch(s2)
  else
    false
  endif
endif
endif
endif
endif
endf
-------------------------------------------------------------------------------------
context State
def MatchS: MatchS(S:State):Boolean =
  FullStateName() = S.FullStateName() and
  entry->forAll(b1:Behavior|S.entry->exists(b2:Behavior|b1.MatchB(b2))) and
  S.entry->forAll(b2:Behavior|self.entry->exists(b1:Behavior|b1.MatchB(b2))) and
  exit->forAll(b1:Behavior|S.exit->exists(b2:Behavior|b1.MatchB(b2))) and
  S.exit->forAll(b2:Behavior|self.exit->exists(b1:Behavior|b1.MatchB(b2))) and
  doActivity->forAll(b1:Behavior|S.doActivity->exists(b2:Behavior|b1.MatchB(b2))) and
  S.doActivity->forAll(b2:Behavior|self.doActivity->exists(b1:Behavior|b1.MatchB(b2))) and
  deferrableTrigger->forAll(t1:Trigger|S.deferrableTrigger->exists(t2:Trigger|t1.MatchT(t2))) and
  S.deferrableTrigger->forAll(t2:Trigger|self.deferrableTrigger->exists(t1:Trigger|t1.MatchT(t2)))
-------------------------------------------------------------------------------------
context ConnectionPointReference
def EqualCPR: EqualCPR(S:ConnectionPointReference):Boolean =
  self.entry->forAll(p1:Pseudostate|S.entry->exists(p2:Pseudostate|p1.EqualPS(p2))) and
  S.entry->forAll(p2:Pseudostate|self.entry->exists(p1:Pseudostate|p1.EqualPS(p2))) and
  self.exit->forAll(p1:Pseudostate|S.exit->exists(p2:Pseudostate|p1.EqualPS(p2))) and
S.exit->forAll(p1:Pseudostate|self.exit->exists(p2:Pseudostate|p1.EqualPS(p2)))

context Vertex
def MatchV: MatchV(V:Vertex):Boolean =
  if self.oclIsTypeOf(Pseudostate) then
    V.oclIsTypeOf(Pseudostate) and (self.oclAsType(Pseudostate).EqualPS(V.oclAsType(Pseudostate)))
  else if self.oclIsTypeOf(ConnectionPointReference) then
    V.oclIsTypeOf(ConnectionPointReference) and (self.oclAsType(ConnectionPointReference).EqualCPR(V.oclAsType(ConnectionPointReference)))
  else false
endif
def allSubRegions: allSubRegions():Set(Region) =
  subvertex->select(oclIsKindOf(State))->iterate(s:State;accRegions:Set(Region)=Set{}|
    if s.submachine->notEmpty() then
      accRegions->union(s.submachine.region)
    else
      accRegions->union(s.region)
    endif
  )
def MatchR: MatchR(R:Region):Boolean =
  let AllSystemVertexes:Set(Vertex) = self.subvertex->union(s:State;accRegions:Set(Region)=Set{}|
    if s.submachine->notEmpty() then
      accRegions->union(s.submachine.region)
    else
      accRegions->union(s.region)
    endif
  )
context Region
def MatchTrans: MatchTrans(T:Transition):Boolean =
  kind = T.kind and
  source.MatchV(T.source) and
  target.MatchV(T.target) and
  guard->forAll(t1:Constraint|T.guard->exists{
    t2:Constraint|t1.specification.stringValue().WCMatch(t2.specification.stringValue())
  }) and
  T.guard->forAll(t2:Constraint|self.guard->exists{
    t1:Constraint|t1.specification.stringValue().WCMatch(t2.specification.stringValue())
  }) and
  trigger->forAll(t1:Trigger|T.trigger->exists(t2:Trigger|t1.MatchT(t2))) and
  T.trigger->forAll(t2:Trigger|self.trigger->exists(t1:Trigger|t1.MatchT(t2))) and
  effect->forAll(b1:Behavior|T.effect->exists(b2:Behavior|b1.MatchB(b2))) and
  T.effect->forAll(b2:Behavior|self.effect->exists(b1:Behavior|b1.MatchB(b2)))

let AllSystemTransitions:Set(Transition) = self.transition
union(self.allSubRegions().transition->asSet()) in
let AllRuleTransitions:Set(Transition) = R.transition
union(R.allSubRegions().transition->asSet()) in
AllRuleVertexes->size()=AllSystemVertexes->size() and
AllRuleTransitions->size()=AllSystemTransitions->size() and
AllSystemVertexes->forall(v1:Vertex|AllRuleVertexes->exists(v2:Vertex|v1.MatchV(v2)))
and
AllSystemTransitions->forall(t1:Transition|AllRuleTransitions->
exists(t2:Transition|t1.MatchTrans(t2)))
------------------------------------------------------------------------
context Class
def ValidateT14: ValidateT14(Sname :String):Boolean =
if self.hasStereotype(Sname) then
let R:Class = Class.allInstances()->any(name=Sname and oclIsTypeOf(Class) and
IsRuleElement()) in
let RRegions:Set(Region) = R.classifierBehavior->select{
oclIsKindOf(StateMachine).oclAsType(StateMachine).region->asSet()} in
let SRegions:Set(Region) = self.classifierBehavior->select(
oclIsKindOf(StateMachine).oclAsType(StateMachine).region->asSet()) in
RRegions->forall(r:Region|SRegions->exists(s:Region|s.MatchR(r)))
else true
endif
--Constraints:----------------------------------------------------------
"}
References


OMG 2003. MDA Guide version 1.0.1 OMG.


An approach for modelling architectural design rules in UML on Object oriented programming, systems, languages, and applications, San Diego, CA, USA 2005 ACM.
Paper III

Communicating architectural design rules using models – An action case study

Anders Mattsson\textsuperscript{1}, Brian Fitzgerald\textsuperscript{2}, Björn Lundell\textsuperscript{3}

1. Combitech AB, Jönköping, Sweden
2. University of Limerick, Limerick, Ireland
3. University of Skövde, Skövde, Sweden

Proceedings of Informing Science & IT Education Conference (InSITE) 2012

Research and writing: Anders Mattsson
Critical revision: Brian Fitzgerald, Björn Lundell
Abstract

An important purpose of architectural design is to ensure that the system meets its quality requirements by defining a set of system wide design decisions. An important part of these design decisions is the set of architectural design rules that shall be followed by developers in the detailed design. The state of practice is to define these rules in natural language and to use manual reviews to enforce them. This way of transferring the rules to the developers is however error prone and requires a lot of effort from the architects since natural language is ambiguous and open for different interpretations and rule following have to be checked with manual reviews. This paper reports from an action case study where a novel approach for architectural modeling and automated conformance checking has been investigated regarding its ability to better communicate architectural design decisions to the developers. The findings indicate that the novel approach is significantly more effective than the state of practice. The findings also show that an important reason for this is that using a tool for conformance checking allows the developers to learn the rules by experimenting.
1 Introduction

An important design artifact in any software development project is the software architecture (Bass, Clements, & Kazman, 2003) consisting of system wide design decisions to be followed in the detailed design. The purpose of the architecture is to guide and control the design of the system so that it meets its quality requirements. Architectural design rules are defined by the architect as a part of the architecture and have to be followed by the developers in their detailed design of the system. A common way of capturing the architecture is to define a high level structure of the system as a set of subsystems with defined interfaces, together with a framework implementing a communication infrastructure used by the components of the subsystems (Mattsson, Lundell, Lings, & Fitzgerald, 2009). This is however not enough; you also need to specify rules as to what kinds of component to put in different subsystems and how the components are supposed to use the infrastructure. These rules cannot be handled by simply putting the components into the subsystems since the rules span more than the current version of the system, so more elements will be added later that also have to follow the rules. Examples of such rules, taken from the embedded software domain, are:

- “A sensor shall be defined in the Sensors package.”
- “A sensor may inherit Infrastructure::Observer to be able to react to changes in Data_Items, for instance to activate or deactivate itself.”
- “A sensor may only have associations to In_Port_IFc and Data_Items. These associations shall only be navigable from the sensor.”
- “A sensor may not have any public operations or attributes.”

Architectural design rules have been recognized as an important part of the architecture from the early works on software architecture by Perry and Wolf (Perry & Wolf, 1992) to recent research in software architecture. In the work of Perry and Wolf architectural design rules are represented as form in the formula: Software Architecture = \{Elements, Form, Rationale\}, where form represents constraints on choice of elements and relationships between elements. In recent research focused on the treatment of architectural design decisions as first class entities (Jansen & Bosch, 2005; Jansen, van der Ven, Avgeriou, & Hammer, 2007; Kruchten, 2004a; Kruchten, Lago, & van Vliet, 2006; Tyree & Akerman, 2005), architectural design decisions impose rules and
constraints on the design together with rationale. Identification of design rules are also typically an activity in methods for architectural design as elaborated below:

- In ADD (Bass et al., 2003; Wojcik et al., 2006) architectural design rules are represented as responsibilities and design constraints.

- In RUP 4+1 Views (Kruchten, 1995, 2004b) architectural design rules are represented as design guidelines.

- In QASAR (Bengtsson & Bosch, 1998; Bosch, 2000; Bosch & Molin, 1999) architectural design rules are represented as rules and constraints.

- In S4V (Hofmeister, Nord, & Soni, 2000; Soni, Nord, & Hofmeister, 1995) architectural design rules are represented as design guidelines and design strategies.

- In ASC (Ran, 2000) architectural design rules are represented as design decisions.

However, neither of these research streams nor methods provides any suggestion on how to model architectural design rules, other than as informal text. There are also numerous languages intended for architectural descriptions, so called Architectural Description Languages (ADL) (Medvidovic, Dashofy, & Taylor, 2007; Medvidovic, Rosenblum, Redmiles, & Robbins, 2002; Medvidovic & Taylor, 2000) (e.g. ACME, Aesop, C2, MetaH, AADL, SysML and UML). However, none of these provide sufficient means to specify constraints or rules on groups of conceptual components only partly specified by the architect where the actual components are intended to be identified and designed by developers in later design phases. The main problem is that the ability to express these kinds of constraints is either missing, or the expressions to define even quite simple rules become much too complicated to be usable in practice. An important factor to remember is that architectural design rules must be easily understandable by architects and developers; otherwise there is a risk that productivity will decrease instead of increase.

As a result of this lack of satisfactory support for modeling of architectural design rules the state of practice is to specify these kinds of rules in informal text. Informal text is ambiguous, open for different interpretations and impossible to automatically enforce. This makes transferring them to the developers an error-prone and time-consuming manual task, as reported in (Mattsson et al., 2009).
There is however a novel approach that promises to solve these problems by providing a method to model architectural design rules and automatically check that the detailed design conforms to the rules (Mattsson, Fitzgerald, Lundell, & Lings, 2012). An important feature of the approach is that the architectural rules are easily understandable by architects and developers, as claimed by the designers of the approach. This paper reports on the findings from an action case study using the approach in an industrial development project. The research objective was to study the effects on productivity and quality as well as the learnability of the approach in order to estimate its effectiveness in transferring architectural design decisions to the developers.

The rest of this paper is organized as follows. In the next section we introduce the novel approach for architectural modeling investigated in the action case study. Thereafter we present the research approach adopted for the study. Following that our findings are presented. Finally we discuss our conclusions.

2 On the investigated approach

In order to be successful in practice, it is essential that architectural design rules are modeled in such a way that they are both amenable to automatic enforcement and easy to understand and use by architects and developers. The latter is important in order to avoid increasing the work of developing the rules; otherwise there is a risk that the work burden is increased instead of decreased even though the enforcement is automated. Another important issue is that it should be possible to use current mainstream modeling tools to model both the architectural design rules and the system model so as to make it widely adoptable.

Such an approach is defined in (Mattsson et al., 2012) with tooling and modeling method available as open source at http://code.google.com/a/eclipselabs.org/p/arcon/. The approach, illustrated in Fig. 1 is based on the idea to use UML (OMG, 2011) to define architectural design rules in a meta-model that constrains the use of UML in a system model.
Fig. 1. An approach for automatic enforcement of architectural design rules.

Classes in the architectural rules model are transformed into UML stereotypes carrying constraints given by the constructs of the architectural rules model. The full mapping between UML constructs in the architectural rules model and the stereotypes used in the system model are provided in Appendix A. Using this approach the constraint, “A sensor may only have associations to In_Port_Ifc and Data_Items. These associations shall only be navigable from the sensor”, is modeled according to Fig. 2. 

Fig. 2. An architectural rule modeled according to the novel approach

Accompanying the method a tool called ArCon (Architectural Conformance Validator) has been built. ArCon is a stand–alone tool that reads the system model and the architectural rules model upon request, and reports on any violations in the system model against the rules defined in the architectural rules model. The idea is that developers should use the ArCon tool regularly to check that they are following the rules, thus eliminating the need for manual architectural reviews.
3 Research approach

3.1 On the research approach adopted

The objective of the research presented in this paper was to understand the effect of using the approach for automatic architectural enforcement illustrated in Fig. 2 in an industrial development project. Of specific interest were effects on productivity and quality as well as the learnability of the method and tool in order to understand its effectiveness in communicating architectural design decisions.

Given this objective, a hybrid of the interventionist/change and interpretivist/understanding perspectives was appropriate. Braa and Vidgen (Braa & Vidgen, 1999) locate a number of hybrid research approaches where a mixture of perspectives is motivating the research, and in case of a mixture of interventionist/change and interpretivist/understanding perspectives, as in this study, the Action Case approach is deemed appropriate. The Action Case approach is a hybrid of the Soft Case and Action Research approaches, each of which is discussed in turn below.

In Soft Case research an interpretivist approach is adopted. The concern is more with gaining understanding and insight (Walsham, 1993). It is our belief that in this area where little exists by way of successful exemplars, then the appropriate approach is an in-depth study which a single case provides, what has been termed the “revelatory case” (Yin, 1994). A single case strategy is also strongly recommended by Mintzberg who poses the very apt question: “what, for example, is wrong with samples of one?” (Mintzberg, 1979). One of the limitations of this study might appear to be the fact that it is based on a single case and thus there is limited scope for generalisation. However, Lee and Baskerville (Lee & Baskerville, 2003) identify a fundamental and long-standing problem with the type of generalization based on the type of statistical sampling frequently sought in research, namely the problem of attempting to generalize to any other settings beyond the current one. Following this conventional model, researchers have suggested increasing sample size or number of case study organizations, but Baskerville and Lee argue cogently for the ultimate futility of this flawed strategy. In terms of external validity, while a single case study cannot achieve statistical generalization(Runeson & Höst, 2009), but rather contribute to an analytical
generalization in which the findings of the case study contribute more generally to developing a theory of the phenomenon being studied.

Action research originates from the work of Lewin (Lewin, 1948) and several ‘flavours’ have emerged. At heart, however, there is general agreement on a number of essential characteristics: it is a highly participative approach which implies a close intertwining between researchers and practitioners intervening on real problems in real contexts, with two primary outcomes: an action outcome in terms of a (hopefully) beneficial intervention in the organisation, and a research outcome in terms of a contribution to research on the phenomenon in question. It is also a longitudinal cyclical process of intervention and reflection, with any learning fed back in successive action research cycles e.g. (Baskerville, 1999; Kock, McQueen, & Scott, 1997; Lau, 1997).

The Action Case is a trade-off between being an observer who can make interpretations (understanding) and a researcher involved in creating change in practice. With case study methods the researcher aims to collect a rich set of data to provide insight into some situation, while in action research the aim is to support desirable change in an organizational setting. However, when doing case studies researchers contribute to change by questioning events and applying new concepts. On the other hand, full-scale action research projects are often not appropriate due to organizational constraints or the nature of the topic to be investigated. Small scale intervention with a deep contextual understanding is one way of balancing this dilemma.

These characteristics were very much present in this research: The intervention was done in a relatively small project over a limited time period, and where the researcher had deep knowledge of the problem domain and the organization. He had 20 years of experience from development of embedded real-time systems across a wide range of organizations in the automotive, defence, medical, telecom and automation industries. The last 15 of these years he had alternated between the roles of a software architect and a mentor in software architecture and Model-Driven Development in numerous projects, several of these in the organization where this action case study was performed. However, in this action case study the role of the researcher was purely that of a researcher, transferring the method to the architect and observing the use of it.
3.2 Data collection and analysis

Data was collected and analyzed following the suggestions of Seaman (Seaman, 1999). Building on constructs derived from earlier work an interview guide was created. Interviews were transcribed and items of relevance were coded. Archival data in the form of models and other development artifacts, as detailed below, was used as a second data source. In order to increase the validity of the results the two data sources were used for triangulation.

Semi-structured interviews were used to provide rich information on the interviewees’ prior knowledge, values and expectations, all important for the estimation of the learnability of the method (Lings & Lundell, 2004). They were also used to provide information on perceived efficiency and quality of the work done.

Two interviews were done with each interviewee. The first, focusing on prior knowledge, values and expectations, was performed before interviewees started to work with the method. The second interview, focusing on usability, learnability of the method, perceived development efficiency and product quality compared to traditional way of working, was performed when they had worked with the method for two to six months.

Four persons were interviewed - the architect and three developers working with the operating system kernel. Each interview started with a specific set of questions prepared in advance. Many of these were open-ended, intended to solicit information not foreseen by the interviewer and encouraging the interviewee to tell context-rich stories. The interviews were recorded and then transcribed. The transcribed interviews were sent to the interviewee who was invited to clarify, correct, and elaborate parts of the interview. All interviewees were also promised anonymity in any reports.

In addition to the interviews archival data was used in the data analysis as a tactic to further minimize the risk of bias from the researcher and improve the validity of the results. The following artifacts produced during development were used as data sources:

- The architectural rules model.
- The system model.
- A text document with the textual rules and additional rationale for the rules
4 Findings

This section describes the research findings, beginning with a description of the research context.

4.1 Action Case Context

The action case study was performed in a development project at a company within the Swedish defense industry. The purpose of the development project was to develop a software platform, including a real-time operating system with special scheduling features, to be used in some of the system computers in a fighter aircraft. The project involved one project manager, one software architect and five developers during sixteen months until first customer delivery.

The project used MDD (Model Driven Development) with Rhapsody as modeling tool. Although there had been previous projects using UML modeling and code synchronization with the Rational Rose tool this was the first project using this tool chain and MDD with full code generation at the company.

The project followed RTCA/DO-178B for level A software, the highest safety level for avionics software. This meant that there were very strict and rigorous rules for how to develop and verify the software. Of specific interest for this study was the fact that although the architectural rules were modeled they still also had to be documented as text. The alternative would have been to qualify the architectural validation tool according to RTCA/DO-178B but that would have been far too costly for this project. The need for keeping the textual rules made it possible to count them and make comparisons between modeled versus non-modeled rules but it also made the benefits of reduced effort for defining the rules less than what could normally be expected.

The project started with an initial phase during one month where the initial architectural rules were modeled in collaboration between the researcher and the architect. Then this architecture model was evolved by the architect alone during four months. At that point a four-hour workshop was held where the rules were presented for the developers by the architect. The developers then started the detailed design according to the architecture. Three incremental iterations were done during twelve months until first delivery.
4.2 Findings in development artifacts

There were 126 rules and 39 guidelines in the Architectural rules document. Rules had to be complied with always, whereas guidelines were recommendations that could be broken if there were good reasons. Guidelines were also often more vague than rules and might require the developer to exercise judgment in order to follow them. Typical examples of rules were: “A Data_Class shall only have private attributes” and “A Data_Class is only allowed to have operations to read and write attributes”. Typical examples of guidelines were “Keep the coupling between components as low as possible” and “The use of inheritance should be avoided”.

Of the 126 rules, 121 were modeled and automatically validated by the tool. The five additional rules could be modeled according to the method but not automatically validated since they were using OCL (OMG, 2003) expressions not supported by the tool. The rules were modeled in seven different class diagrams. The most complex one is shown in Fig. 3.

Of the 39 guidelines 18 could have been modeled and automatically validated by the tool. However, the architect chose not to model guidelines at all. The motivation for this was that since following them was not mandatory, there was a risk of ‘polluting’ the validation reports with error messages that should have been treated as warnings. The remaining 11 guidelines that could not be modeled were all of the nature of requiring the developer to exercise judgment in following them, which made them inherently impossible to formalize.

Thus, in total 73% of the rules and guidelines could be automatically verified and potentially, with some additions to the tooling, 89% would be possible to automate. This would have a major impact both on reduced effort for manual reviews and on the quality of the system.
4.3 Effects on productivity

The architect stated that even though he was forced to develop both textual rules and the model, due to process requirements (see section "Error! Reference source not found."), he gained efficiency in developing the architecture. This was due to the fact that the model was at a higher level of abstraction than the textual rules, making the complexity lower, and that the formalism of the models eliminated ambiguous and contradicting rules. It was easier, more efficient and gave better results to first model the rules and then manually extract the textual rules from the model, than the traditional practice to work with rules only as text. He estimated that in a normal situation where the textual rules would not be needed he would gain between 10 to 50% depending on the complexity of the rules, in this phase. The greatest effect for his productivity though, he estimated, was that he had to spend a lot less time communicating and reviewing the design than in a normal MDD project. Normally he would spend one to two days doing an architectural review; this time was now reduced to less than an hour checking the five rules not checked by the tool. He also used to spend a large part of his time explaining the meaning of the rules to the teams, with this new way of working there was almost no such questions. Spending a lot less time on reviews and communicating the rules enabled him to do optimizations to the architecture and supporting the teams in their design which, he stated, had made the system better and the teams more efficient.

All developers claimed that they found it faster to read and understand the modeled rules than the textual rules. The only difficulty they saw was that the error messages from the tool sometimes were difficult to understand, but as one of them pointed out:
“This is the same problem you have with a new compiler; the error messages can be hard to understand at the beginning but you soon learn how to interpret them”.

All developers believed that after a few days they were already more efficient then they would have been if working the traditional way, and were sure that the quality, in respect to how they had followed the rules, was much higher than if they would have had to manually follow the textual rules. This, they estimated, had an even greater impact on their productivity since there was no longer any rework after the architectural reviews.

4.4 Effects on quality

According to the architect, when working in the traditional way where the architectural rules are enforced by manual reviews, generally a few violations are not detected until very late in the process, typically in the design of a future version of the system, since many architectural rules address maintainability, scalability and portability requirements. With automatic detection of all violations there are no violations to the architectural design rules in the system. According to Boehm and Basili (Boehm & Basili, 2001):

“Finding and fixing a software problem after delivery is often 100 times more expensive than finding and fixing it during the requirements and design phase.”

This means that eliminating architectural violations should have a major impact on the quality of the system. The architect also claimed that the rules were of higher quality, compared to textual rules, since there were no ambiguous, redundant or contradicting rules due to the formal modeling.

Another effect was that both the review and the discussions about the architecture between the architect and the developers focused on how to package the functionality in different components and not on how to follow detailed design rules since this latter question was automatically handled. The architect claimed that in a “normal” project this would have been the opposite way around and that he believed that the increased
focus on more difficult design decisions also led to a better system. The only apparent risk was that of relying too much on the tool and missing things that were not checked by the tool either by design or because of possible errors in the tool.

4.5 Learnability of the method

The architect had eleven years of experience from UML modeling working as an architect in several organizations within the embedded software industry. He had also quite deep knowledge of the UML meta-model from building model-to-code transformations in MDWorkbench. The architect was positively disposed to using the approach since he was the one who had taken the initiative to use it after having seen a presentation of the method and tool at an internal workshop arranged for technology leaders within the company.

Developer A had a master degree in mechatronics and automation and two years of experience of design and implementation of embedded software. Developer B had a bachelor degree in electronics and three years of experience in design and implementation of embedded software. Developer C had a master degree in software engineering and electronics and two-and-a-half years of experience of design and implementation of embedded software. None of the developers had any knowledge of meta-modeling and all had only a one-week training course in UML modeling. All developers were positively disposed to using the method and tool based on expectations that it would reduce the effort needed to follow the architectural rules.

The architect and a researcher familiar with the method collaborated in developing an initial version of the architectural rules model with an effort of about two person-weeks during one month. After that the architectural rules model was evolved by the architect alone during four months with an effort of about three person-weeks. At that point the model was presented to the developers in a four-hour long workshop. After that there were only minor adjustments to the architectural rules model. The architect estimated that it took about one month to master most of the modeling concepts and an additional month to fully master the method.
Both the developers and the architect emphasized the benefits of getting immediate feedback from the tool, it made the learning of the rules much faster. All three developers claimed that they found it easy to understand the modeled rules right from the beginning with the primary reasons being that the rules were modeled in UML, which they all had some experience of, and that they got instant feedback from the validation tool, illustrated by these statements:

- “If you understand UML then you understand the rules”
- “Instant feedback is the only efficient way to learn. If you give a banana to a monkey one hour after it has done something good nothing happens, if you do it instantly, it will learn to do it again”

5 Conclusions

There is a lack of satisfactory solutions on how to model architectural design rules in the current body of literature. As a result the state of practice is to specify these kinds of rules in informal text. Informal text is ambiguous, open for different interpretations and impossible to automatically enforce. This makes transferring them to the developers an error-prone and time-consuming manual task. The objective of this study was to understand the effects of using a novel approach that promises to solve this problem in an industrial development project.

Several findings indicate that the approach provides a more efficient way of communicating architectural design decisions than the traditional way:

- Normally the architect would spend one to two days doing an architectural review; this time was now reduced to less than an hour checking the five rules not checked by the tool. The architect also used to spend a large part of his time explaining the meaning of the rules to the teams, with this new way of working there was almost no such questions.
- All developers claimed that they found it faster to read and understand the modeled rules than the textual rules.
- All developers claimed that they found it easy to understand the modeled rules right from the beginning.
- The architect claimed that the rules were of higher quality, compared to textual rules, since there were no ambiguous, redundant or contradicting rules due to the formal modeling.
Both the review and the discussions about the architecture between the architect and the developers focused on how to package the functionality in different components and not on how to follow detailed design rules since this latter question was automatically handled. The architect claimed that in a “normal” project this would have been the opposite way around and that he believed that the increased focus on more difficult design decisions also led to a better system.

An important reason for the more efficient communication was that the rules were modeled in UML, which is a widely spread modeling language that all interviewees had experience in. Another important reason, as pointed out by several of the interviewees, was that they got instant feedback from the validation tool whether they followed the rules or not.

Also the approach proved to be easy to learn, especially for the developers. The demands are of course much higher for the architect, who is tasked with constructing the rules, so it is not surprising it takes longer to master the method, still, two months must be considered to be a relatively short time to fully master a new modeling technique. It probably takes longer without any knowledge of meta-modeling but judging from how fast the developers, without any meta-modeling experience, understood the models this should not be a major hurdle. A lack of experience in UML modeling (or more significantly, any OO modeling language) would probably be a bigger problem, but UML modeling are today to be considered to be a basic required skill for a software architect.

However, since change in work practice requires people to change, perhaps the most important finding was the enthusiasm expressed both by the architect and the developers, they all thought that they produced a better result with less effort and had more fun doing it. As expressed by one of the interviewees:

- “I think you have come up with something really useful here.”

Although the action case study only covered one development project, two factors suggest that the results should, to a large extent, be transferable to other systems and organizations, at least in the embedded software domain:
1. The defined transformations are based on raising the general modeling constructs of UML to the meta-model level, not on the specific needs of the developed system.

2. The developed system is a real-world system.

References


Kruchten, P. (2004a). An ontology of architectural design decisions in software intensive systems, 2nd Groningen Workshop on Software Variability (pp. 54-61).


**Appendix I**

In Table 1 the definition on how to interpret the constructs in the architectural rules model as constraints on a system model is given. The definitions refer to an architectural rules model complying with the form of the generic model given in Fig. 4. References to terms defined in the generic model are written in italics.
Fig. 4. A generic architectural rules model used in the definition of the transformations

Table 1. Definition of transformations between constructs in the architectural rules model and constraints on stereotypes in the system model

<table>
<thead>
<tr>
<th>Transformation</th>
<th>Architectural rules model</th>
<th>System model stereotypes</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1: A class named C1 with the stereotype M1 is transformed into a stereotype named C1 extending the meta-class M1 unless transformation number T3 below applies. If M1 is undefined then “Class” is assumed</td>
<td>&lt;&lt;Package&gt;&gt; Sensor_Pkg</td>
<td>UML::Package UML::Class</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Sensor</td>
</tr>
<tr>
<td></td>
<td>&lt;&lt;own.attribute&gt;&gt; SamplingPeriod</td>
<td></td>
</tr>
<tr>
<td>T2: If SR2 is the role in the language meta-model on the far end of an association from the meta-class of C1 to the meta-class of C2 then the multiplicity of R2 for a &lt;&lt;C2&gt;&gt; element shall be constrained to Mn2 in stereotype &lt;&lt;C1&gt;&gt;</td>
<td>&lt;&lt;Class&gt;&gt; Sensor</td>
<td>&lt;&lt;stereotype&gt;&gt; Sensor_Pkg [&lt;&lt; stereotype &gt;&gt; Sensor ]</td>
</tr>
<tr>
<td></td>
<td>&lt;&lt;own.attribute&gt;&gt; SamplingPeriod</td>
<td></td>
</tr>
<tr>
<td>T3: If M1 equals “metaclass” then C1 represents the class C1 in the UML meta-model and is not transformed into anything in the system model. This can be used to specify constraints in other stereotypes in respect to these meta-classes</td>
<td>&lt;&lt;Class&gt;&gt; Sensor</td>
<td>&lt;&lt;stereotype&gt;&gt; Sensor</td>
</tr>
<tr>
<td></td>
<td>&lt;&lt;own.attribute&gt;&gt; SamplingPeriod</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;&lt;own.attribute&gt;&gt; Property</td>
<td></td>
</tr>
<tr>
<td>T4: If SA equals “meta” far an attribute A and the name of A matches the name of an attribute of class M1 in the meta-model then it is transformed into a constraint on that attribute on allowed values. The value of the attribute is constrained to match a regular expression specified as the default value of the attribute.</td>
<td>&lt;&lt;Property&gt;&gt; SamplingPeriod</td>
<td>&lt;&lt;stereotype&gt;&gt; SamplingPeriod</td>
</tr>
<tr>
<td></td>
<td>&lt;&lt;meta&gt;&gt; name = &quot;SamplingPeriod&quot; &lt;&lt;meta&gt;&gt; visibility = “private”</td>
<td></td>
</tr>
<tr>
<td>T5: If no match is found for an A where SA equals “meta” (according to T4) then A is transformed into an attribute A of the stereotype (tag-definition), thus defining a tagged value to be set in the model elements where the stereotype is applied</td>
<td>&lt;&lt;Class&gt;&gt; All_Classes</td>
<td>&lt;&lt;stereotype&gt;&gt; All_Classes</td>
</tr>
<tr>
<td></td>
<td>&lt;&lt;meta&gt;&gt; Designer: String</td>
<td>Designer: String</td>
</tr>
<tr>
<td>Transformation</td>
<td>Example</td>
<td></td>
</tr>
<tr>
<td>---------------</td>
<td>---------</td>
<td></td>
</tr>
<tr>
<td><strong>Architectural rules model</strong></td>
<td><strong>System model stereotypes</strong></td>
<td></td>
</tr>
<tr>
<td>T6: Any OCL constraint in the context of a class $C_1$ is copied exactly as it is into the stereotype $C_1$.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| ```ocl
{ inv: self.base.Class.ownedOperation.
  extension_Sample.size()=1 xor
  self.base.Class.ownedOperation.
  extension_Trig.size()=1 }
``` | ```ocl
{ inv: self.base.Class.ownedOperation.
  extension_Sample.size()=1 xor
  self.base.Class.ownedOperation.
  extension_Trig.size()=1 }
``` |
| T7: A generalisation relationship from a class $C_3$ to a class $C_1$ in the architectural rules model is transformed to a generalisation from stereotype <<$C_3$>> to stereotype <<$C_1$>>. The UML meaning of this is that all constraints and attributes of the stereotype <<$C_1$>> are inherited by the stereotype <<$C_3$>> and that any <<$C_3$>> element is also an <<$C_1$>> element. | ![Diagram](image1) | ![Diagram](image2) |
| T8: If $M_1$ equals "Package" and the aggregation of $R_1$ is "composite": A <<$C_1$>> Package is constrained to have $M_2$ number of <<$C_2$>> elements as packagedElements. The visibility of these elements shall be the visibility of $M_2$. Also, a <<$C_1$>> package is not allowed to have any packagedElements unless explicitly allowed in the model. | ![Diagram](image3) | ![Diagram](image4) |
| T9: <<$C_1$>> elements are only allowed to have the associations, dependencies, generalizations and realizations explicitly allowed according to T10 and T11. | See examples for T10 and T11. |
| T10: If $M_A$ equals "Association": A <<$C_1$>> element shall be associated with $M_2$ number of <<$C_2$>> elements. The association ends shall have the same navigability, aggregation (none, shared or composite) and visibility as $R_1$ and $R_2$. The association ends shall also have qualifiers according to the qualifiers of $R_1$ and $R_2$. The name and type of these shall be according to the transformations for attributes specified in T12. The association shall have the stereotypes $S_1$ to $S_n$. | ![Diagram](image5) | ![Diagram](image6) |
## Transformation

<table>
<thead>
<tr>
<th>T11: If MA equals “Dependency”, “Generalization” or “Realization” and the association is only navigable from C1 to C2:</th>
</tr>
</thead>
<tbody>
<tr>
<td>A &lt;&lt;C1&gt;&gt; element shall have a relationship according to MA to Mu2 number of &lt;&lt;C2&gt;&gt; elements with stereotypes S1 to Sn</td>
</tr>
</tbody>
</table>

**Architectural rules model**

```
<<Realization>>
In_Port
```

**System model stereotypes**

```
<< stereotype >>
In_Port
(An << In_Port >> Class shall realize one << In_Port_ifc >> Class)
```

<table>
<thead>
<tr>
<th>T12: If there are attributes A of C1 where SA is not equal to “meta”:</th>
</tr>
</thead>
<tbody>
<tr>
<td>All parts of the definition of an attribute of a &lt;&lt;C1&gt;&gt; class must match the corresponding part of an A, where the wild card characters “@” and “%” in any part of the definition of A can be replaced with any character sequence. Parts of A not specified (as for instance default value for Sampling_Period in the example to the right)</td>
</tr>
<tr>
<td>All A must be matched by one attribute in a &lt;&lt;C1&gt;&gt; class. An exception to this is if the name of A contains the wild card character “%”; in this case any number of matches (including zero) is allowed.</td>
</tr>
<tr>
<td>If the name of a type of A is identical to the name of a class C in the architectural rules model then the type of a matching attribute must be a &lt;&lt;C&gt;&gt; element.</td>
</tr>
</tbody>
</table>

**Architectural rules model**

```
<<Class>>
Sensor
- Sampling_Period : int
- % : @
```

**System model stereotypes**

```
<< stereotype >>
Sensor
(A << Sensor >> Class must have one private attribute named Sampling_Period with a type named int and any number of other private attributes with any type)
```
## Transformation

<table>
<thead>
<tr>
<th>Architectural rules model</th>
<th>System model stereotypes</th>
</tr>
</thead>
</table>
| **T13:** If there are operations O of C1:  
All parts of the definition of an operation of a <<C1>> class must match the corresponding part of an O, where, for each part of the definition, the wild card characters “@” and “%” can be replaced with any character sequence. Properties of O not specified (as for instance parameter directions for operations in the example to the right) are unconstrained.  
This requirement holds for all parts of the definition of O defined in the UML meta-model, such as for instance opaque behaviour specified for the operation.  
The character “%” in a parameter name means that the definition of this parameter can be repeated any number of times, including zero. In these parameter definitions “%” can be replaced with any character sequence.  
If the name of the type of O or a parameter of O is identical to the name of a class B in the architectural rules model then the type of matching operations or parameters in the <<C1>> class must be of a <<B>> Class.  
All O must be matched by one operation in a <<C1>> class. An exception to this is if the name of O contains the wild card character “%”; in this case any number of matches (including zero) is allowed.  
A <<Subject>> Class shall have one public operation named ‘Attach’ with one parameter named “O”. The type of this parameter shall be a Class stereotyped <<Observer>>. The operation shall have no return value | **<<stereotype>>**  
**Subject**  
(A <<Subject>> Class shall have one public operation named ‘Attach’ with one parameter named “O”. The type of this parameter shall be a Class stereotyped <<Observer>>. The operation shall have no return value) |
| ![Diagram](Subject.png)  
**Subject**  
+Attach(O:Observer)  
**Data_Item**  
+Set_%(%:@)
  | **<<stereotype>>**  
**Data_Item**  
(A <<Data_Item>> Class shall have any number of public operations who's name begins with "Set_" . The operations can have any parameters of any type. The operations shall have no return value. The operations shall have an opaque behaviour specification ending with 'Notify();') |
| ![Diagram](Data_Item.png)  
**Data_Item**  
+Set_%(%:@)  
  | **<<stereotype>>**  
**Sensor**  
(A <<Sensor>> Class shall have a state machine with a top level region with a state machine that is a copy of the state machine of Sensor class in the architectural rules model) |
| ![Diagram](Sensor.png)  
**Sensor**  
Sampling  
after(Sampling_Period):Sample()  
  | **<<stereotype>>**  
**Sensor**  
(A <<Sensor>> Class shall have a state machine with a top level region with a state machine that is a copy of the state machine of Sensor class in the architectural rules model) |

## Example

- **Subject**
  
  +Attach(O:Observer)
  
  **Data_Item**
  
  +Set_%(%:@)
  
  ```
  Subject
  +Attach(O:Observer)
  
  Data_Item
  +Set_%(%:@)
  
  Set_%(%:@)
  |
  @
  Notify();
  ```

- **Sensor**
  
  ```
  Sensor
  Sampling
  after(Sampling_Period):Sample()
  ```

- **Data_Item**
  
  ```
  Data_Item
  +Set_%(%:@)
  
  Set_%(%:@)
  |
  @
  Notify();
  ```

- **Subject**
  
  ```
  Subject
  +Attach(O:Observer)
  ```

- **Sensor**
  
  ```
  Sensor
  Sampling
  after(Sampling_Period):Sample()
  ```

- **Data_Item**
  
  ```
  Data_Item
  +Set_%(%:@)
  
  Set_%(%:@)
  |
  @
  Notify();
  ```
Bibliography and References


Liskov, B. and Zilles, S. (1974) 'Programming with abstract data types', in Proceedings of the ACM SIGPLAN symposium on Very high level languages, Santa Monica, California, United States, ACM,


Zdun, U. and Avgeriou, P. (2005) 'Modeling architectural patterns using architectural primitives', in 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications, San Diego, CA, USA, ACM,