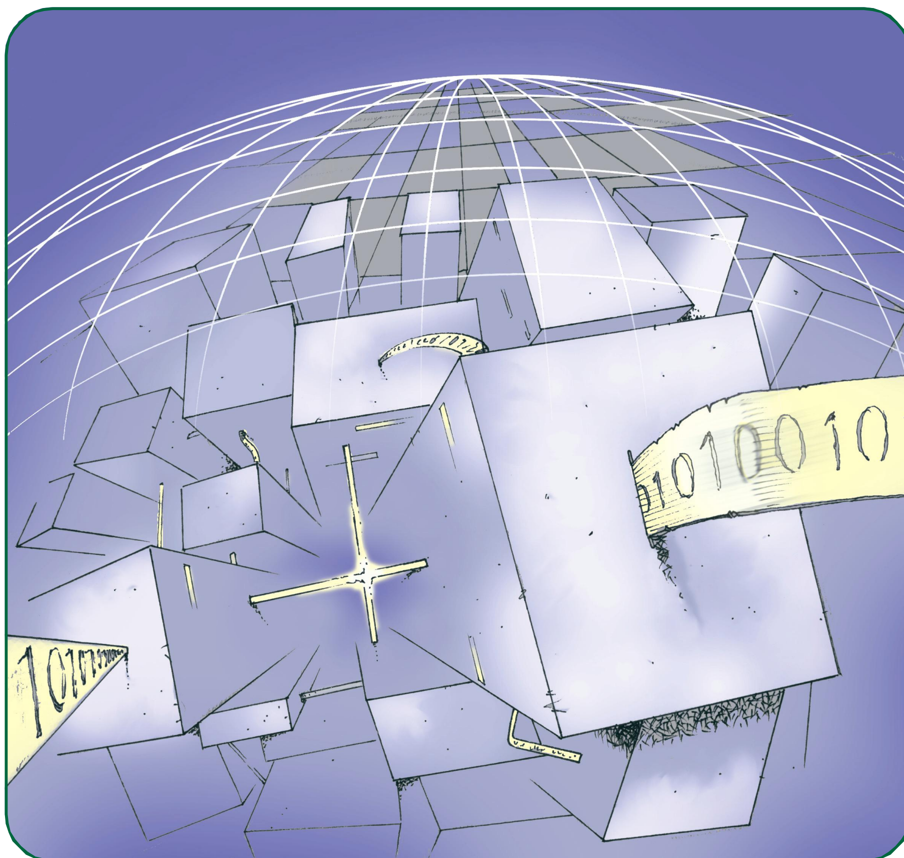
ESPOO 2005 VTT PUBLICATIONS 561



Janne Merilinna

A Tool for Quality-Driven Architecture Model Transformation

VTT

VTT PUBLICATIONS 561

A Tool for Quality-Driven Architecture Model Transformation

Janne Merilinna

VTT Electronics

ISBN 951–38–6439–1 (soft back ed.)
ISSN 1235–0621 (soft back ed.)

ISBN 951–38–6440–5 (URL: http://www.vtt.fi/inf/pdf/)
ISSN 1455–0849 (URL: http://www.vtt.fi/inf/pdf/)

Copyright © VTT Technical Research Centre of Finland 2005

JULKAISIJA – UTGIVARE – PUBLISHER

VTT, Vuorimiehentie 5, PL 2000, 02044 VTT
puh. vaihde 020 722 111, faksi 020 722 4374

VTT, Bergsmansvägen 5, PB 2000, 02044 VTT
tel. växel 020 722 111, fax 020 722 4374

VTT Technical Research Centre of Finland, Vuorimiehentie 5, P.O.Box 2000, FI–02044 VTT, Finland
phone internat. +358 20 722 111, fax + 358 20 722 4374

VTT Elektroniikka, Kaitoväylä 1, PL 1100, 90571 OULU
puh. vaihde 020 722 111, faksi 020 722 2320

VTT Elektronik, Kaitoväylä 1, PB 1100, 90571 ULEÅBORG
tel. växel 020 722 111, fax 020 722 2320

VTT Electronics, Kaitoväylä 1, P.O.Box 1100, FI–90571 OULU, Finland
phone internat. +358 20 722 111, fax +358 20 722 2320

Technical editing Anni Kääriäinen

Otamedia Oy, Espoo 2005

Merilinna, Janne. A Tool for Quality-Driven Architecture Model Transformation [Työkalu arkkitehtuurimallin laatuohjattuun transformaatioon]. Espoo 2005. VTT Publications 561. 106 p. + app. 7 p.

Keywords model-driven development, Model-Driven Architecture

Abstract

Model-Driven Development (MDD) is about treating models as first class design entities. Model-Driven Architecture (MDA) is an Object Management Group's initiative that proposes to define a set of non-proprietary standards that will specify interoperable technologies with which to realize MDD with automated transformations. The concept of Model-Driven Architecture lies on models at different abstraction levels, where transformations are performed switching between models. Transformations where the abstraction level is changed are called vertical transformations to separate from horizontal transformations where the abstraction level remains unchanged.

Quality-driven model transformation is a horizontal transformation where varying quality attributes of a software product are the driving force for transformation. The quality-driven model transformation relies on the fact that the functionality of the system can be implemented with a wide variety of architectures and therefore with different quality properties. The purpose is to conform to the MDA approach and thus, the goal is to automate the transformation with advanced CASE (Computer Aided Software Engineering Tool) tool.

This thesis focuses on designing and implementing a tool extension that automates the quality-driven model transformation. To accomplish this, a rule description language for defining transformation rules was developed. In addition, a CASE tool evaluation was performed to find the most suitable modelling tool to be extended. Finally, the tool extension was implemented to the Telelogic Tau/Developer.

Merilinna, Janne. A Tool for Quality-Driven Architecture Model Transformation [Työkalu arkkitehtuurimallin laatuohjattuun transformaatioon]. Espoo 2005. VTT Publications 561. 106 s. + liitt. 7 s.

Avaisnsanat model-driven development, Model-Driven Architecture

Tiivistelmä

Malliohjatun kehittämisen ajatuksena on käyttää malleja ensisijaisina suunnittelukohteina. Model-Driven Architecture (MDA) on Object Management Groupin ehdotus kehittää yleishyödyllisiä standardeja, jotka määrittelisivät keskenään yhteensopivia teknologioita, joita voitaisiin käyttää malliohjatun kehittämisen toteuttamiseen automaattisilla transformaatioilla. MDA:n perusajatus on käyttää eri abstraktiotasoilla olevia malleja, joissa mallista toiseen voidaan liikkua tekemällä transformaatioita. Transformaatioita, joissa abstraktiotasoa vaihdetaan, kutsutaan vertikaalisiksi transformaatioiksi ja transformaatioita, joissa abstraktiotaso ei muutu, kutsutaan horisontaalisiksi transformaatioiksi.

Laatuohjatun mallin transformaatio on horisontaalinen transformaatio, jossa ohjelmistotuotteen muuttuvat laatuvaatimukset ovat transformaation peruste. Laatuohjattu mallin transformaatio perustuu siihen tosiseikkaan, että järjestelmän toiminta voidaan toteuttaa monella eri arkkitehtuurilla ja täten eri laatuvaatimuksilla. Tarkoituksena on pyrkiä noudattamaan MDA-lähestymistapaa, joten päämääränä on automatisoida transformaatio CASE-työkalun avulla.

Tämän lopputyön tavoitteena oli kehittää työkalulaajennus, joka toteuttaa laatuohjatun mallin transformaation. Tavoitteen saavuttamiseksi kehitimme transformaatioiden kuvaamista varten sääntökuvauskielen. Lisäksi teimme mallinnustyökaluvertailun, jonka tavoitteena oli löytää sopiva työkalu laajennusta varten. Lopuksi toteutimme työkalulaajennuksen Telelogic Tau/Developeriin.

Preface

This thesis was completed within the VTT Electronics Software architecture group. Work for this was carried out under Families (FAct-based Maturity through Istitutionalisation Lessons-learned and Involved Exploration of System-family engineering) project under Eureka S! 2023 Programme, ITEA project ip02009.

I would like to thank Scientist Mari Matinlassi and Research Professor Eila Niemelä from VTT Electronics for guiding me through this work. I would also like to thank my supervisors Professors Tapio Seppänen and Jukka Riekki from the University of Oulu from their constructive criticism.

Oulu, January 24, 2005

Janne Merilinna

Contents

Abstract ... 3

Tiivistelmä ... 4

Preface ... 5

Abbreviations and acronyms ... 9

1. Introduction ... 11

2. Software architecture development ... 14
   - 2.1 Quality-driven Architecture Development ... 14
   - 2.2 Unified Modeling Language ... 19
   - 2.3 Model-Driven Architecture ... 22
     - 2.3.1 The Model ... 24
     - 2.3.2 Abstraction Levels ... 24
     - 2.3.3 Platform ... 25
     - 2.3.4 Model Transformations ... 26

3. Quality-driven model transformation ... 30
   - 3.1 Overview of the Technique ... 31
   - 3.2 Quality-driven Rule Description Language ... 33
   - 3.3 Applying Quality-Driven Model Transformation ... 36
     - 3.3.1 Applying the Stylebase ... 36
     - 3.3.2 Applying Admissibility Rules ... 38
     - 3.3.3 Defining Mappings ... 39
     - 3.3.4 Defining Rules by Q-RDL ... 42
     - 3.3.5 Performing Layers-to-Blackboard Transformation ... 45

4. Evaluation of UML tools for model-driven architecture ... 47
   - 4.1 The First Tool Evaluation – Literature Study ... 48
   - 4.2 The Second Tool Evaluation – Empirical Study ... 50
   - 4.3 Summary ... 53

5. Development of the Q-Tra tool ... 55
   - 5.1 Requirements for the Tool Extension ... 55

5.1.1 End-User Requirements 56
5.1.2 Modelling Tool Requirements 57
5.1.3 Technical Requirements 57
5.2 Design of the Q-Tra Tool Extension 58
5.2.1 Technical Constraints for Designing the Q-Tra 59
5.2.2 Architecture of the Q-Tra 61
5.3 Implementation of the Q-Tra Tool Extension 76
5.3.1 Implementation of the Components 76
5.3.2 Testing the Components 85

6. Case study – layers-to-blackboard transformation 87

7. Discussion 94
7.1 Experiences in Applying Quality-driven Rule Description Language 95
7.2 Analysis of the Tool Evaluation Result 96
7.2.1 Experiences of Using Telelogic Tau/Developer 97
7.3 Future Development of the Q-Tra 99
7.3.1 Databases 99
7.3.2 User Interface 100
7.3.3 Modelling Tool 101

8. Summary 102

References 104

Appendices
Appendix 1: The Q-RDL in Extended Backus-Naur Form
Appendix 2: Contents of the Rulebase
Appendix 3: Contents of the Stylebase

Abbreviations and acronyms

3GL	3rd Generation Language, for example C++ and Java
4GL	4th Generation Language, for example SQL
CASE	Computer Aided Software Engineering, use of computer-based support in the software development process
CIM	Computation Independent Model, abstraction level of Model-Driven Architecture
CORBA	Common Object Request Broker Architecture, middleware technology
DiSeP	Distribution Service Platform, platform for software components
DTD	Document Type Definition, defines legal building blocks of an Extensible Mark-up Language document
EBNF	Extended Backus-Naur Form, context-free grammar
GUI	Graphical User Interface, graphical interface for the user to interact with a computing system
IEEE	Institute of Electrical and Electronics Engineers, electronic library
ISO	International Standardization Organization
J2EE	Java 2 Enterprise Edition, standard for developing component-based multitier enterprise applications
MDA	Model-Driven Architecture, framework for standards that will enable model-driven development
MDD	Model-Driven Development, software development method
MSMQ	Microsoft Message Queuing, messaging infrastructure
OMG	Object Management Group, standardization organization for object-based technologies
PF	Product Family, family of products

PFA	Product Family Architecture, software structure that is common for all products of a product family
PIM	Platform Independent Model, abstraction level of Model-Driven Architecture
PSI	Platform Specific Implementation, abstraction level of Model-Driven Architecture
PSM	Platform Specific Model, abstraction level of Model-Driven Architecture
QADA	Quality-driven architecture design and quality analysis, architectural design and analysis method
Q-RDL	Quality-driven Rule Description Language, transformation rule description language
Q-TRA	Quality-driven architecture TRAnsformation, tool that automates quality-driven architecture model transformation
SQL	Structure Query Language, language for accessing databases
UML	Unified Modeling Language, object-based modelling technology
WSDL	Web Service Definition Language, grammar for describing network services
XML	Extensible Mark-up Language, information representation language

1. Introduction

Model-Driven Development (MDD) is about treating models as first class design entities. Modelling provides a view to a complex problem and its solutions, which is less risky, cheaper and easier to understand than implementation of the genuine target. [1]

Model-Driven Architecture (MDA) is defined as “an OMG initiative that proposes to define a set of non-proprietary standards that will specify interoperable technologies with which to realize model-driven development with automated transformations” [2]. The concept of Model-Driven Architecture lies on three types of models on the different abstraction levels: computation independent model (CIM), platform independent model (PIM) and platform specific model (PSM). The computation independent model shows the system in the environment where it will operate. The platform independent model concentrates on the operation of the system while hiding the details of the underlying platform. PIM is computationally complete meaning that it is possible to execute the system defined by this model. The platform specific model is described as a realization of PIM with all the details of the chosen platform. [3]

Model transformation is described as “the process of converting one model to another model of the same system” [2]. Transformations where the abstraction level is changed are called vertical transformations to separate from horizontal transformations where the abstraction level remains unchanged. Horizontal transformation is used when models are enhanced, filtered and specialized during the design process [4].

In the sense of MDA, quality-driven model transformation is described as a PIM-to-PIM transformation where varying quality attributes are the driving force and the reason for the transformation [5]. Quality-driven model transformation relies on the fact that the functionality of the system can be implemented with wide variety of architectures and therefore with different kind of quality properties, such as performance, modifiability and extensibility.

From the point of view of MDA, transformation from PIM into the desired PSM is essential when considering run-time properties, but sometimes it is not

enough. In order to change evolution time qualities, i.e. modifiability and extensibility, it is necessary to change the architecture of the system in the PIM level to correspond to the new quality requirements.

Benefits from automating quality-driven model transformation are quite self-explanatory. An architect can easily experiment and try different kinds of architectures for a system while designing a model just by a press of a button, when traditionally every change in the model has to be done manually. Particularly in the context of product families, automated quality-driven model transformation is justified.

The software product family is a family of products sharing a set of common properties and architecture – product family architecture (PFA). However, products of a product family may have various customer groups desiring different qualities from a product. For instance, for one customer the hard real-time requirements are essential but for another, reliability is important. Automated quality-driven model transformation enables easy optimization or change of desired quality property of a product.

Quality-driven model transformation is based on the quality-driven model transformation technique [5]. The technique aims at conforming to the MDA approach and therefore its goal is to automate the transformation with advanced Computer Aided Software Engineering (CASE) tools.

The aim of this thesis is to develop a tool that automates the quality-driven model transformation. In order to accomplish this, the following steps have to be taken:

- to develop a rule description language, which describes the transformation rules defined by the quality-driven model transformation technique
- to find the most suitable CASE tool for the tool extension
- to design and implement the tool extension.

This thesis is structured as follows: First, the background information related to the quality-driven model transformation is introduced. Second, the quality-driven model transformation technique is introduced briefly. In addition, the

Quality-driven Rule Description Language (Q-RDL) is presented. Q-RDL is applied to present the transformation rules defined by the technique. Furthermore, an example transformation of applying the Q-RDL is given. Third, evaluation of UML modelling tools in the context of MDA is performed. This part of work has been published in the paper "Evaluation of UML Tools For Model-Driven Architecture" [6]. The result of the evaluation is a modelling tool that is extended to support quality-driven model transformation. Fourth, the design and implementation of the tool extension, the Q-Tra, is introduced. In order to validate the automated quality-driven model transformation, a simple case study where the Q-Tra is applied is presented next. Finally, the experiences of the Q-RDL and the tool evaluation and the Q-Tra are discussed and some future improvements are introduced.

2. Software architecture development

This section defines the background information related to the quality-driven model transformation. First, quality-driven architecture development is introduced by bringing out the basic terminology of the quality properties of software architectures. Second, Unified Modeling Language is introduced. Third, model-driven development and its realization, Model-Driven Architecture, are discussed in order to get basic knowledge of modelling and model transformations.

2.1 Quality-driven Architecture Development

Software architecture is described as a structure or structures of the system. Structures consist of the software components and their externally visible properties, and relationships among them. [7] Bosch presents three purposes of an explicit representation of the software architecture [8]:

- stakeholder communication
- software product lines
- quality attributes assessment.

The first reason for the explicit software architecture is that it allows early communication between stakeholders involved in the development process. The development process cannot proceed until the stakeholders have accepted the architecture.

The second reason for the explicit software architecture is that it defines components in the software product line. A software product line is a group of systems that share common software architecture and a set of reusable components. A software product line and software product family are often considered synonyms, but there are some distinctions. A software product line is considered more a process approach of making software products and thus emphasizes inputs and outputs of the development process. A software product family is a product oriented term emphasizing – in addition to process and architecture – also the business and organizational aspects of a product family [9].

The product family architecture is an architecture, which is derived of products of a product family. As the product family members share the same architecture and for the fact that the architecture constrains quality attributes of a system, choosing the right architecture is essential. This occurs especially in the case of PFA, as the quality attributes reflect on the whole product family.

Quality attributes are non-functional features of a system that are often divided into two main categories [10]:

- Execution qualities, i.e. performance, availability, reliability, etc.
- Evolution qualities, i.e. maintainability, modifiability, reusability, etc.

Execution qualities are discernible at run-time and evolution qualities are considered in the architecture development. For example, quality attributes can be defined as follows:

- *Availability* measures the proportion of time the system is up and running.
- *Extensibility* is the systems' capability to acquire new components.
- *Maintainability* is the ease with which a software system or component can be modified or adapt to a changed environment.
- *Modifiability* is the capability of making changes quickly and cost-effectively.
- *Portability* is the capability of the system of running under different computing systems.
- *Reliability* is a system or component capability of keeping operating over the time or of performing its required functions under stated conditions for a specific period of time.

There are four concepts of software architecture that must be defined:

- *The architecture style* is a description of component types and their topology. A style defines constraints on the architecture and the constraints define a set of architectures that satisfies them. Thus, architecture style is not architecture, but it still conveys an image of the system. [7]

- When the architecture style is strictly defined, it becomes an *architecture pattern*. The architecture pattern expresses fundamental structural schema for software systems, which are applied for high-level system subdivisions, distribution, interaction and adaptation. [11]
- *The design pattern* describes a recurring structure of communication components, which solves a general problem in a particular context. [12] As design patterns are applied in a particular context, they can be considered micro architectures.
- *Idioms* are programming language specific design patterns. Thus, they are the lowest level patterns.

The basic principle of quality-driven architecture development is to emphasize the importance of quality attributes at the development time. That is, designing software architecture with specific patterns and thus with specific quality attributes. Applying QADA® (Quality-driven architecture design and analysis) [13] method is one approach of designing software architectures from quality point of view. The Layers and the Blackboard architecture patterns are described as examples of architecture patterns and the quality attributes they promote.

According to Buschmann [11], the Layers architectural pattern "helps to structure applications that can be decomposed into the groups of tasks, in which each group is at a particular level of abstraction". A well-known architectural model, which implements Layers architectural pattern, is OSI 7-Layer Model, defined by the International Standardization Organization (ISO).

The bearing idea in Layers is that each layer only communicates with the layer below and thus hides the implementation of the lower layer from layers above. A layer includes an inner structure, and thus, it has several kinds of internal components. Components at higher layers may communicate with the components of the layer below directly or via an interface object. Figure 1 presents the structure of the Layers architectural pattern. The components in Figure 1 could also be arbitrary but the structure is the same.

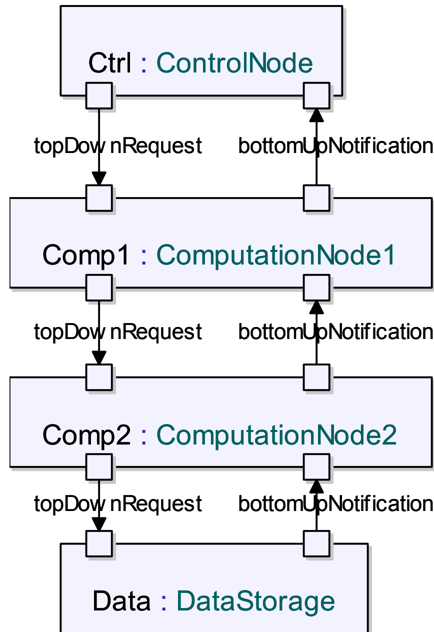


Figure 1. Layers structure.

The quality attributes Layers promotes stems from the fact that the layers of the pattern can easily be replaced. For quality attributes of the Layers architecture pattern, Niemelä et al. defines four quality attributes [14]:

- maintainability
- modifiability
- portability
- reusability.

According to Buschmann [11], the Blackboard pattern "is useful for problems for which no deterministic solution strategies are known" and "in Blackboard several specialized subsystems assemble their knowledge to build a possibly partial or approximate solution".

The idea behind Blackboard is to have a collection of independent components to work cooperatively on a common data structure. The blackboard is a central data store, where all knowledge sources have access. The knowledge sources are independent subsystems that solve some specific aspects of the overall problem.

The component, which organizes the whole system, is called Control. Control component evaluates the current state of processing and coordinates the knowledge sources. Figure 2 illustrates the structure of the Blackboard architectural pattern.

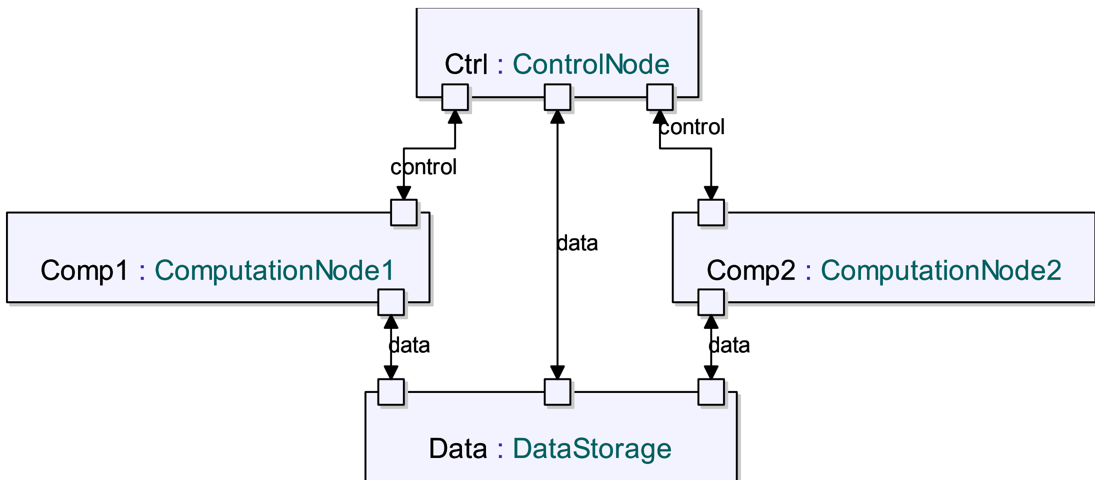


Figure 2. Blackboard structure.

Niemelä et al. defines five quality attributes for the Blackboard architecture pattern [14]:

- availability
- maintainability
- modifiability
- reliability
- reusability.

Maintainability, modifiability and reusability stems from the Blackboard's capability of allowing even dynamic addition and removal of the components. Availability and reliability originates from the fact that the computation components are independent from each other and the control component iteratively activates the other components. Therefore, a fault in one component may not cause complete failure of the system if the control component handles exceptions successfully.

2.2 Unified Modeling Language

Unified Modeling Language (UML) is Object Management Group's (OMG) standardized graphical modelling language for specifying, visualizing and constructing, and documenting software systems [15]. Modelling is performed using several different diagrams to express the system's high-level behaviour, static and dynamic structure, and dynamic behaviour.

Current (November 2004) official UML version 1.5 offers nine different diagrams for specifying the system's behaviour and structure. *Use case* diagrams are mainly used for expressing requirements of the system, *class* and *object diagrams* are used for describing static structure and *component* and *deployment diagrams* catch the system's implementation structure. Behaviour is modelled with *communication*, *sequence*, *state chart* and *activity diagrams*. [16] Each diagram type focuses on certain aspects of the system only. However, it is not necessary, or even the intention, to use all the diagrams when designing a system.

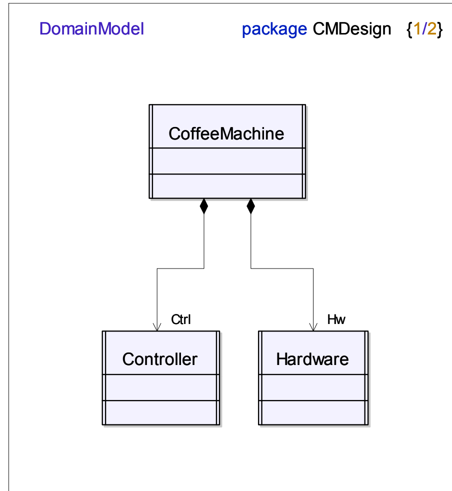
Although UML has a rich collection of diagrams and the intention is to use just a subset of the diagrams in the development process, there are some flaws and weaknesses in its designing capabilities [16]. From the point of expressing architectural, structural and real-time aspects, some capabilities are missing. For instance, real-time system consists of various independent processes that communicate with themselves. This kind of communication, e.g. which part is communicating with which part and with what kind of signals, is generally expressed with ports, which are attached to active objects. Ports are connected together by communication channels that allow sending and receiving different kind of signal and finally, protocols, which coordinate whole I/O-activity. Regardless, UML 1.5 does not support these kinds of expressions. There are some means, for instance *sequence* or *collaboration diagrams* and *class diagrams* for expressing communication structure, but in the end, they are not sufficient. Besides, expressing protocols is not even possible.

By the time of writing (November 2004) this thesis, the current adopted UML version is 1.5, but in the near future version 2.0 will be published. Perhaps the most significant addition to this new version from the perspective of describing software architecture [17], will be a better support to express software decomposition, e.g. expressing internal structure of classes and components [18].

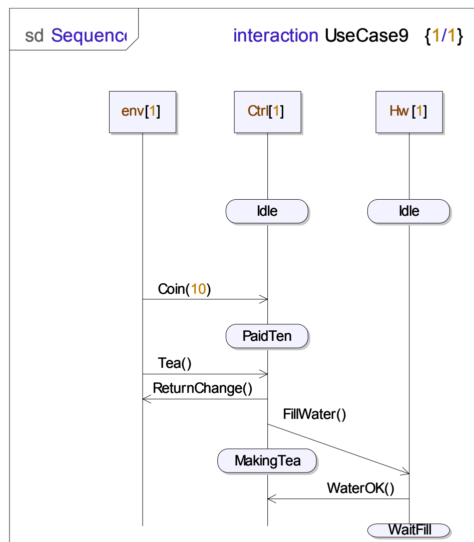
While in UML 1.5, classes' and components' internal structure could only be described with *class diagram*, which however cannot be used for describing hierarchical structure of models, UML 2.0 offers a completely new diagram for the task. *Composite* s*tructure diagram* describes how a containing element is composed by other elements called *parts*, which are not themselves classifiers, such as classes, but more like instances of classifiers, and their communication paths. These *parts* can also have their own inner structure as well and for that reason, expressing a model in the desired abstraction level is possible. Communication paths between *parts* are described by using connectors. Connectors are connected to the *ports*, which also show interfaces of the component that can be accessed. [19]

With *composite structure diagram*, it is possible to describe the classes' inner structure and its *parts* interaction quite easily compared to the language UML 1.5 provides. Figure 3.a and 3.b describe an example of the means UML 1.5 provides for expressing communication structures and Figure 4 shows the means UML 2.0 will provide.

In Figure 3.a, CoffeeMachine class composes of two sub classes: Controller and Hardware. Figure 3.b tries to show the communication between sub classes, but the *sequence diagram* only displays one use case and for that reason, it does not show all the signals that can be sent and received and thus interfaces between classes remain undefined.



a) Class diagram.



b) Sequence diagram.

Figure 3. Communication structures of UML 1.5.

In Figure 4, there are two *parts* named Ctrl, which is the instance of Controller, and Hw, which is the instance of Hardware, inside of active class CoffeeMachine. Ctrl and Hw communicate with each other through a *port*, which is connected together by a *connector*. The *Interfaces* between *parts* can be

seen close by the *ports*. Ctrl *provides interface* for CoffeeOK, WaterOK and Warm signals and *requires interface* for FillCoffee, FillWater and HeatWater signals. The *provided interface* is defined as an interface of services that the component offers, provides, for the other components. The *required interface* is defined as an interface of services, which has to be *provided* by the other components. Ctrl is also communicating with the environment of the active class CoffeeMachine through the *port* of the CoffeeMachine. With this notation, it is easy to describe the composition and communication of the classes.

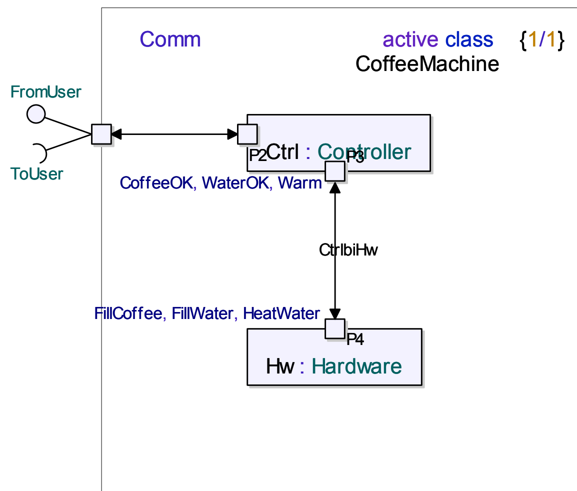


Figure 4. Composite structure diagram.

UML 2.0 introduces new ways for modifying and extending the actual UML itself. *Profiles* are used to extend UML with domain specific elements. For instance, the UML profile for real-time systems might include some extra information for the model about timing, performance, scheduling policies, etc. [16]. In this way, the language itself is not overloaded with all the features that are not needed in every software system.

2.3 Model-Driven Architecture

Modelling in traditional engineering is considered essential or even compulsory. No one would ever begin to build a new car or an air plain without first constructing a proper model of it. However, in software business, modelling is

quite seldom used and when applied, models are left to play a secondary role. This is peculiar, as software systems are highly complex nowadays and the benefits of using models and modelling techniques could be considerable. [1]

Model-Driven Development is about treating models as first class design entities [1]. While traditionally models end up just as documentation and, in addition, they are far too often inconsistent with the source code, in MDD, the whole source code is to be generated from models. However, both the software modelling and code generations had been tried for years with quite limited success and mostly in highly specialized domains, but until now standards and automation technologies have been quite immature.

Model-Driven Architecture is defined as "an OMG initiative that proposes to define a set of non-proprietary standards that will specify interoperable technologies with which to realize model-driven development with automated transformations". [2] In addition to the MDD's approach of using modelling languages as programming languages, Model-Driven Architecture also tries to solve a problem that troubles the software business: varying software platforms and technologies. To solve this, system functions have to be defined in completely platform independent fashion. This has also been tried before with a wide variety of different kinds of middleware solutions, such as Common Object Request Broker Architecture (CORBA), but results have been quite heterogeneous. In MDA, the idea is to use multiple models at different levels of abstraction to isolate the system specification from the underlying platform. In this way, the platform heterogeneity is hidden at the design- and compilation-time.

The promise of MDA is to solve or ease the "hot new technology" effect. This is possible as the system is modelled in a platform independent fashion and for that reason, existing designs can be targeted to new implementation infrastructures. Maintenance should also be much easier as the availability of design in machine readable-form gives direct access to the specifications of the system. Since the source code can be generated from developed models, testing and simulation of the system is made much easier. The models can be validated against requirements, tested against various infrastructures and the actual behaviour of the system can be simulated in the very beginning of the software development. [3]

To get a better grasp of MDA, a set of central concepts has to be defined and explained. First, the very basics of MDA are discussed by defining what the actual *model* is and then by defining what *abstraction levels* there are in MDA, starting from the most abstract one and ending up to the actual implementation. Finally, it is explained how *transformations* are used to switch between *models*. However, no too specific issues are discussed here, only the necessary aspects for understanding the central thinking behind MDA.

2.3.1 The Model

The model is described as a simplified representation of the system. The model does not answer all questions about the system. It only answers a subset of them from whose point of the view it is made. For instance, a globe is a model of the Earth. Distances between countries and continents can be pieced together when knowing the scale. However, it is not possible to tell the temperature of some place on the Earth by just looking at the globe. For this reason, multiple *views* of the system usually exist to answer different kinds of questions. For instance, the weather model tells everything about the weather. The actual *view* is defined as follows: "A view is a model that is completely derived from another model (the base model). A view cannot be modified separately from the model from which it is derived. Changes to the base model cause corresponding changes to the view. If changes are permitted to the view, then they modify directly the source model." [20] The model can be described either in textual or graphical language with strictly defined syntax and semantics.

2.3.2 Abstraction Levels

The concepts of Model-Driven Architecture lies on three different types of models: a computation independent model, a platform independent model and a platform specific model.

The computation independent model is a model, which shows the system in the environment where it will operate. None too specific details of the system are presented, as typically this model is independent of how the system is

implemented. For this reason, CIM is sometimes called business or domain model. [3]

The platform independent model concentrates on the operation of the system while hiding the details of the underlying platform. This model is computationally complete meaning that it must be possible to execute the system defined by this model. A platform independent model does not change from one to another when changing the implementation platform. [3]

The platform specific model can be described as a realization of PIM with all the details of the chosen platform. For example, a CORBA specific PSM could be expressed in the UML profile for CORBA and a Web Services PSM could be expressed in Web Services Description Language (WSDL). [21] The actual source code can also be considered as a PSM but sometimes it is called Platform Specific Implementation (PSI) to separate it from graphical presentations.

2.3.3 Platform

”A platform is a set of subsystems and technologies that provide a coherent set of functionality through interfaces and specified usage patterns, which any application supported by that platform can use without concern for the details of how the functionality provided by the platform is implemented.” [3] In this thesis, the term 'platform' is used in the same way as David Frankel in the book Model Driven Architecture [22]:

- information-formatting technologies, such as XML DTD and XML Schema
- 3GLs and 4GLs, such as Java, C++, C# and Visual Basic
- distributed component middleware, such as J2EE, CORBA and .NET
- messaging middleware, such as MQSeries and MSMQ.

When platform independency and platform independent or specific models are discussed, platform has to be defined to make sense in definitions. Especially when speaking of a platform independent model. Figure 5 represents the definition dilemma. The communication middleware PIM transforms into CORBA-specific model, or briefly into a platform specific model. At the same

time it is PSM for the communication middleware and PIM for the operating system. It is the same case with, for example, Java. Java is usually considered platform independent but in fact, it relies strictly on its virtual machine. This is why defining the platform is crucial.

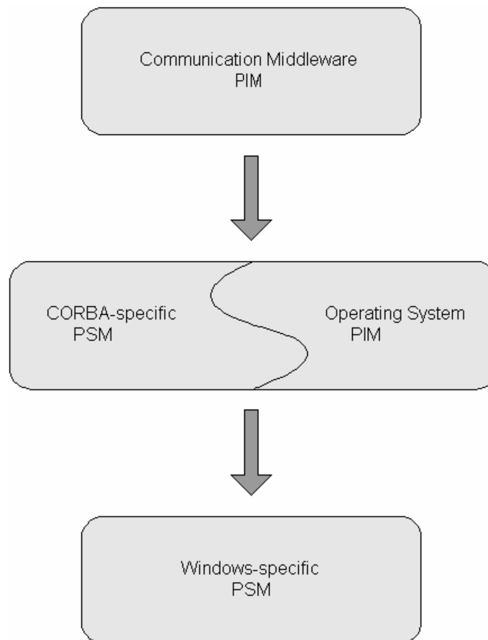


Figure 5. Platform independency is relative.

2.3.4 Model Transformations

Model transformation is described as "the process of converting one model to another model of the same system" [3]. In MDA, one of the key transformations is from PIM to PSM, but also several other transformations are defined. Transformations where the abstraction level is changed are called vertical transformations to separate from horizontal transformations where the abstraction level remains unchanged [23]. Next, vertical transformations from CIM to PIM, PIM to PSM and PSM to the code are explained briefly:

- CIM to PIM. This transformation might be somewhat abstract as computation independent, or business, models are typically not

appropriate to express all the details needed for PIM. For this reason, PIM may be drawn separately, but all the requirements and other aspects defined in CIM are taken into consideration.

- PIM to PSM. This transformation is used when the PIM is sufficiently defined and its function is secured. In this transformation, the platform specific issues are attached to the PIM to form PSM, which should then be completely aware of its platform.
- PSM to code. This transformation is used when all the platform specific details are defined and the model is ready for actual implementation.

Horizontal transformation, that is, transformation between models at the same abstraction level, is also possible. For an example, PIM to PIM transformation is used when models are enhanced, filtered and specialized during the design process. [4]

However, vertical transformations can be considered the key transformations, as they push the system under development from specifications all the way to the actual source code. Figure 6 represents the key idea of the MDA. As at the top there is a platform independent model which is obviously does not rely on any platform, it can be transformed into any desired target platform and finally to the running code. This is the method, how the isolation between specification of the system and the implementation is achieved.

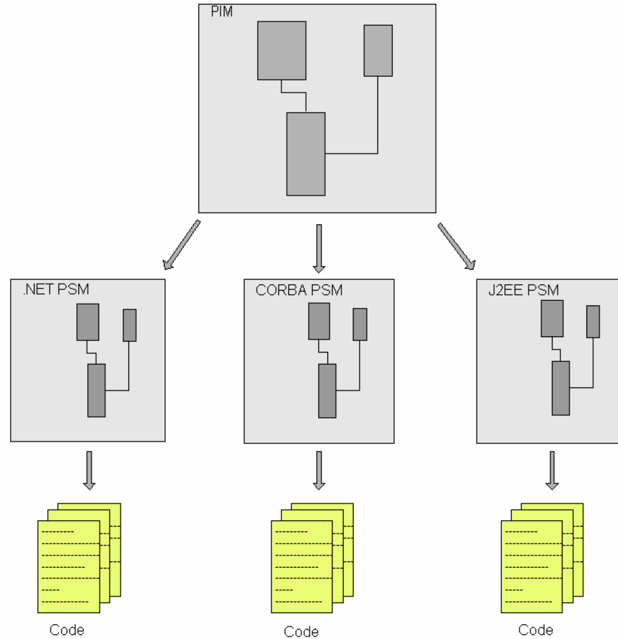


Figure 6. PIM transforming into various PSMs.

Transformations are based on *mappings*, which are defined as a specification of a mechanism for transforming the elements of a source model to the elements of a target model [3]. *Mapping* makes use of things called *marks*, which are attached to the model and used to guide the transformation. However, if no user-defined marks are attached to the model, transformation can be considered deterministic. That is, all the information required for transformation is encapsulated in the source model. In this way, a certain source model always transforms into a certain target model.

Either the transformation is vertical or horizontal; it should be possible to transform the model back to the original model from where it was transformed. This is where the *transformation record* is considered. When the transformation is conducted, the result is the target model, for instance from PIM to PSM transformation it is PSM and *transformation record.* The record of transformation includes a map between elements of the original model to the target model [3]. With the use of the *transformation record* it is possible to trace every aspect and requirement defined in CIM all the way to the actual implementation and vice versa.

If the *transformation record* is discarded, undoing the transformation is made difficult. This is especially the case, when the abstraction level is changed. Consider for instance, a PSM-PIM-PSM transformation. First, the platform specific model, in this case the source code, is transformed into an implementation language independent model, PIM. In the code, there are loops implemented with three kinds of ways: *while-*, *do-while-* and with *for*-loops. How do we express distinction between these loops in PIM? Furthermore, when the PIM is transformed back to PSM, does the original PSM, the source code, match perfectly with the transformed (target) PSM. Particularly if the optimization has taken place in any stage, it is highly unlikely that the models do match.

A similar problem is encountered even if the abstraction level is not changed. For instance, a simple PIM-PIM transformation where all the classes' public attributes are changed to private. There is no way to undo the transformation if there is no record available.

If all or even some of the transformations can be automated, the benefits are quite self-explanatory. Automated vertical transformations reduce time to market, increase productivity and may also increase quality of the product, as well defined transformations and code generation may produce better quality of the source code than by hand writing it. The situation is the same, when C++ is first compiled for assembly. It takes an expert programmer to produce a better assembly code than a modern compiler does. Again, benefits of using horizontal transformations might result in a more qualified model.

3. Quality-driven model transformation

A quality-driven model transformation [5] can be described in the sense of MDA as a PIM-to-PIM transformation. This means that the transformation is performed between models at the same abstraction level and, in this case, between two platform independent models.

The quality-driven model transformation is justified especially in the context of product families. As stated in Section 2.1, the products of a product family share the same architecture and common properties. However, products of the same family may promote different kinds of quality requirements, but they share the same functionality. For instance, one product has to be as reliable as possible, but for another, hard real-time requirements, i.e. performance is essential.

The driving force for taking quality-driven model transformations is in the varying software quality requirements. The change may result from varying software platforms and middleware, change in underlying hardware or domain standards. Alteration in the quality requirements of a software product requires modifications either in the behaviour or structure or in both of these. In the context of quality-driven model transformation, variations of architecture models are within the scope.

Quality-driven model transformation relies on the fact that the functionality of the system can be implemented with a wide variety of architectures and with different quality attributes in mind. From the point of view of MDA, transformation from PIM into the desired the PSM is essential, but sometimes it is not enough to develop new products if the platform, whether it is software or just plain hardware, or the quality requirements of the product differs a lot from quality the product family supports. Sometimes it is necessary to change the architecture of the system first in the PIM level to correspond to the new quality requirements and after that, the transformation from PIM to PSM can be conducted.

Designing software architecture is about constructing a high-level structure for the software systems. Diverse *architecture styles* or *patterns* are most suitable for certain situations. Implementing the software with wrong architecture may result in wide variety of problems. On the other hand, choosing the correct

architecture from the start will result in – from some point of view – more qualified model.

Knowing the benefits of using a certain pattern in a certain situation will help avoiding pitfalls while constructing the ultimate structure – architecture – for the software systems. For this purpose, there are widely available catalogues concerning *design* and *architecture patterns*, e.g. [11] and [12]. At the end, choosing the correct architecture means gathering information about the problem domain, prioritising requirements – functional and non-functional – and making design decisions.

3.1 Overview of the Technique

Quality-driven model transformation technique defines the following inputs [5]:

- What information is required to make transformation possible?
- Where does the information stand?
- How is the information obtained and used?

An overview of the technique is described in Figure 7. At the top of Figure 7 stands the source model, which is to be transformed to the other (target) model with different quality attributes. The architect determines which architecture is the best solution for the architecture of the system by considering the information gathered in special database called the stylebase.

The stylebase contains a set of design patterns and architectural styles or patterns [14]. The idea is to gather all the information of the patterns in a uniform way to promote automation of the technique as much as possible. The following 12 aspects are stored in the stylebase: name of the pattern, reference, quality attributes, component types, component roles, connector types, data topology, control topology, purpose, diagram name, abstraction level and optional rationale [5]. With the information gathered of each pattern, especially quality-attributes, the architect can consider the most suitable design or architectural patterns for the system.

Transformations are defined by the following the rules defined for the transformations. The rules consist of the transformation *admissibility rules* and *feasibility rules* [5] which, combined and considered together, are used to form *mappings*. *Mappings* set the discipline how the transformation from a source model to a target model can be implemented.

It seems that applying the quality-driven model transformation technique for defining new transformations results in more pattern pair-specific rules than the general rules of the transformations. In order to make automating transformations possible, all the transformation *mappings* have to be defined explicitly. For this purpose, we created a special database called rulebase, which contains all the pair-specific *mappings* defined in a uniform way.

In order to pinpoint components from a model and to conduct actual transformation, the source model has to be *marked* first. Matinlassi declares three *marks* that have to be attached to the components of the source model: the name of the pattern the component participates in, the component's role in it and the component's type [5]. The actual transformation can then be conducted by considering the *marks* of the model and by applying *mappings* found in the rulebase.

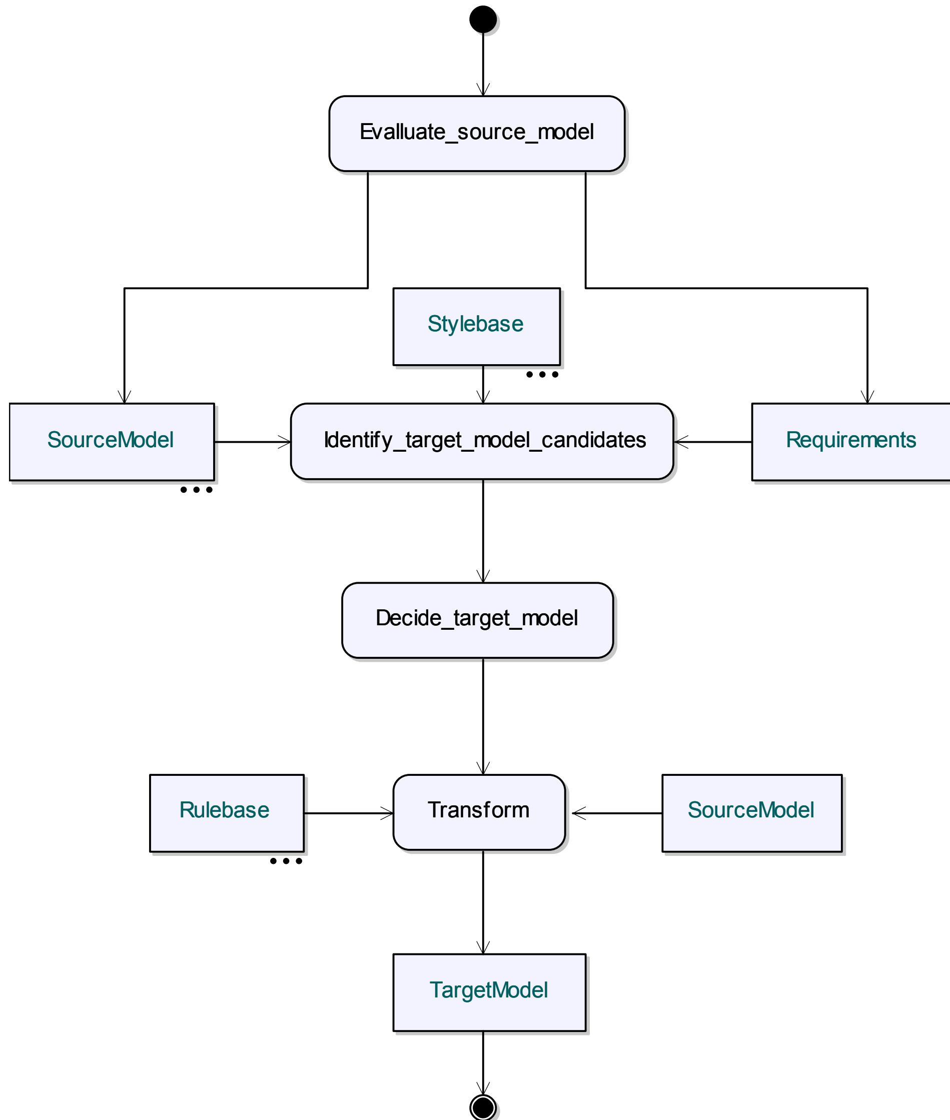


Figure 7. An overview of the quality-driven model transformation technique.

3.2 Quality-driven Rule Description Language

The best way to define transformation rules is to apply a standard transformation description language. Object Management Group announced a request for proposals [24] in April 2002 and initial submissions were due to October 28th 2002. A total of eight proposals was received. A review of all proposals is

available in [20]. Currently (November 2004), transformation description languages are still underway and thus we decided to invent our own rule description language.

The quality-driven model transformation technique promotes automation by suggesting that the *mappings* between pattern-pairs are presented in a uniform way. For this purpose, we introduce Quality-driven Rule Description Language (Q-RDL), which is used to describe the *mappings* defined by the technique. It must be emphasised that Q-RDL is not a general-purpose rule description language and it can only be applied, as such, to describe transformation rules defined by the quality-driven model transformation technique. It must also be understood that the Q-RDL is still an immature rule description language.

Mappings described with Q-RDL for transformation includes the following parameters:

1. Source pattern name
2. Target pattern name
3. Source component marks
    a. Pattern name
    b. Component role
    c. Component type
4. Target component marks
    a. Pattern name
    b. Component role
    c. Component type
5. Crucial component marks
    a. Pattern name
    b. Component role
    c. Component type
6. Connection rules
    a. Source component marks
        i. Pattern name
        ii. Component role
        iii. Component type
    b. Target component marks
        i. Pattern name
        ii. Component role
        iii. Component type

The first two parameters on the list present source and target pattern names. These are only specified once at the top of every transformation rule.

Parameters 3 and 4, which are source and target component marks, concern *mark mappings* between the patterns. This means that every source component with certain list of *marks*, which are pattern name, component role and component type, is mapped to the corresponding component with target *marks*. Source and target component marks always exist in pairs and there may be an unlimited number of these pairs in all transformation rules.

The fifth parameter, crucial components marks, consists of a list of components, which are necessary to the target pattern. This information is needed if some components are missing in the transformation from the source to target model, which are necessary for the target pattern to function. If this happens, the fifth parameter proposes creating a new component into the target model with the desired *marks*.

The sixth parameter consists of connection rules between components. Here we have the source and target component *marks*, which provide information on what kind of components are to be connected together. It must be noticed that 'source' does not refer to the source model but to the source component in the target model and similarly 'target' refers to the target component in the target model.

All parameters and attributes are presented in a single list, which follows the same order as the parameters presented above. The parameters are separated from each other by descriptive tags. The syntax of the Q-RDL is described in Extended Backus-Naur Form (EBNF) in Appendix 1. Appendix 2 represents an example transformation from Layers to Blackboard pattern defined in Q-RDL.

Defining quality-driven model transformation with Q-RDL works well, if the transformations can be defined as one-to-one transformations, as the pair-specific rules indicates in the first place. The one-to-one transformation means that transformation from the source to target pattern can be described explicitly and that the particular pattern is always transformed in the same way to this other pattern. However, this is not the case with all transformations. Transformations exist where the source is successfully transformed into some

particular target model, but reverse transformation is not possible without user interaction. There is this defect certainly in the Q-RDL but the actual transformation can be questioned in these kinds of situations. There is no available technique, which could automate one-to-many, many-to-one and many-to-many transformations without user interaction.

3.3 Applying Quality-Driven Model Transformation

Here, we illustrate the quality-driven model transformation technique with a simple transformation example. The example illustrates how the Layers architectural pattern can be successfully transformed into the Blackboard pattern. The purpose of this example is to show how to

- apply the stylebase for defining source and target patterns
- apply the admissibility rules for validating transformation admissibility
- define transformation mappings
- apply Q-RDL for rule description
- perform the Layers-to-Blackboard transformation by marking the source model and applying Q-RDL.

3.3.1 Applying the Stylebase

The stylebase requires the following information of each pattern [5]: name of the pattern, reference, quality attributes, component types, component roles, connector types, data topology, control topology, purpose, diagram name, abstraction level and optional rationale.

The first pattern we insert to the stylebase is called Layers. For the name of the pattern we can set “Layers” and for reference we set “Bushmann, F., et al., Pattern-oriented software architecture – a system of patterns, 1996, Chichester, New York: Wiley” to make sure which “Layers” pattern we are defining. For the quality attributes, Niemelä [14] defines four quality-attributes: portability, modifiability, maintainability and reusability. However, maintainability can be defined as a composite of the other quality attributes [10] such as flexibility,

reusability, modifiability, testability and integrability and therefore maintainability is a too abstract definition for the stylebase. In addition, maintainability is rather a system level attribute than an architecture level attribute. For that reason, all the quality-attributes except maintainability are added to the stylebase. Inside Layers, there may be different types of components and no specific types are presented, thus we set "varying" to the stylebase. It is similar with the component roles. We set "layer" and "component" to the stylebase. Communication between components and layers are often called top-down requests if the communication propagates from top to down, and bottom-up notifications in a contrary situation. Thus, we set "top-down requests" and "bottom-up notifications" into the stylebase concerning connector type parameters. Data and control topologies conforms hierarchical topology [25], for this reason "hierarchical" is set to both data and control topology parameters in stylebase. For the purpose, Buschmann [11] defines "from mud to structure". As Layers is concerned with how the components relate and communicate with each other, a UML 2.0 diagram where the pattern exists is set to "composite structure" diagram [18]. The Layers being an architectural pattern, the abstraction level is set to "architectural" and rationale is left blank, as no comments for the pattern are set at this time. Table 1 summarises the Layers architectural pattern.

Table 1. Layers information.

Stylebase parameter	Value
Name	Layers
Reference	[11]
Quality attribute	Portability, modifiability, reusability
Component type	Varying
Component role	Layer, Component
Connector type	Top-down request, bottom-up notification
Data topology	Hierarchical
Control topology	Hierarchical
Purpose	From mud to structure
Diagram	Composite structure
Abstraction level	Architectural
Rationale	

The second pattern we insert to the stylebase is called Blackboard. The required information is gathered in the same way as for Layers. In addition to the quality attributes Niemelä [14] defines, extensibility is added to the stylebase as it can be considered that Blackboard promotes extensibility, by providing loose coupling between components and thus adding new components to the system may require only minor modifications to the data and control components. Extensibility is defined a systems capability to acquire new components. [10] However, the quality attributes maintainability is left out again. Table 2 summarises the Blackboard architectural pattern.

Table 2. Blackboard information.

Stylebase parameter	Value
Name	Blackboard
Reference	[11]
Quality attribute	Reliability, modifiability, reusability, extensibility, availability
Component type	Data, control, computation
Component role	Blackboard, control, source
Connector type	Messages
Data topology	Hierarchical
Control topology	Star
Purpose	From mud to structure
Diagram	Composite structure
Abstraction level	Architectural
Rationale	

3.3.2 Applying Admissibility Rules

Now, we have defined how the source and the target patterns are defined in the stylebase. To ensure transformation admissibility, we apply admissibility rules [5] to the patterns. The first rule states that transformation is admissible only between patterns at the same level of abstraction. If we compare Table 1 and Table 2, the outcome is true. The second admissibility rule states that the purpose of the patterns has to be the same. This is also true. As both the admissibility rules are true, we can move on.

3.3.3 Defining Mappings

In order to develop *mark mappings* between the Layers and the Blackboard patterns, we have to approach the *mapping* problem by constructing a simple example, which should describe at least the most of possible *mappings. Mappings* developed by using this example could be used later on in all Layers-to-Blackboard transformations.

Since component types are not pre-defined in the Layers patterns, we have to set some types for them to make it possible to form *mark mappings*. The first feasibility rule [5] states that the component type is not allowed to vary in the transformation. In Blackboard, data, control and computation components exist, and thus we set these types of components into the model, which illustrates the Layers pattern (see Figure 1) to form an example *mark mappings* without special exceptions.

In addition, the corresponding roles of the components are set for both models. For the Layers model, we set "layer" and "component" roles for the components. For the Blackboard model, we set corresponding roles of the types of the components. Table 3 presents the *marks* of the source model and Table 4 presents the *marks* attached to the target model.

Table 3. Marks of the Layers model.

Component name	Pattern name	Component role	Component type
Ctrl	Layers	Component	Control
Comp1	Layers	Component	Computation
Comp2	Layers	Component	Computation
Data	Layers	Layer	Data

Table 4. Marks of the Blackboard model.

Component name	Pattern name	Component role	Component type
Ctrl	Blackboard	Control	Control
Comp1	Blackboard	Source	Computation
Comp2	Blackboard	Source	Computation
Data	Blackboard	Blackboard	Data

Now, we have set *marks* for both the source and target models. *Mark mappings* can be easily formed by just replacing the source pattern marks with the corresponding target pattern marks. The component type will not change in transformation. Table 5 presents *mark mappings* between Layers and Blackboard architectural patterns. Here, the *marks* of the source components are replaced with *marks* of the target component.

Table 5. Mark mappings between Layers and Blackboard.

Mapping pairs	Pattern name	Component role	Component type
Source component	Layers	Layer	Data
Target component	Blackboard	Blackboard	Data
Source component	Layers	Component	Control
Target component	Blackboard	Control	Control
Source component	Layers	Component	Computation
Target component	Blackboard	Source	Computation

In practice, *mark mappings* mean that the *marks* of the source components (Table 3) are changed to corresponding *marks* of the target components (Table 4).

In this stage, it is reasonable to consider what happens if some components are missing in the source model that are crucial for the target model. The first feasibility rule points out that the type of a component is not allowed to change during transformation, thus all the component types of the target pattern have to exist in the source model. If all these types of components do not exist in the source model, they are generated during the transformations to the target model.

In this way, the transformation is made possible, but considering the reasonableness of the transformation, it is left for the shoulders of the architect. The list of the crucial components is specific for each pattern, not for the transformation. For instance, when transforming (from an arbitrary pattern) to Blackboard, the list of crucial components is always control, computation and data. *Marks* of the crucial components are collected in the table (Table 6).

Table 6. Marks of the crucial components.

Crucial component	**Pattern name**	**Component role**	**Component type**
1.	Blackboard	Control	Control
2.	Blackboard	Source	Computation
3.	Blackboard	Blackboard	Data

The second feasibility rule [5] states that the connector topology is constructed from the scratch and thus the connection rules depend solely on the target pattern. Considering Blackboard (see Figure 2) reveals three kinds of rules for the Blackboard pattern:

- Computation components have access to data components.
- All computation components are controlled by a controller component.
- A control component has access to the data component.

These rules are then collected in the table. In the Table 7, the source component is connected to the target component. Both the components are identified by using *marks*. That is, all the components with certain *marks* (source component) are always connected to the components with specific *marks* (target component).

Table 7. Connector mappings.

Connection pairs	Pattern name	Component role	Component type
Source component	Blackboard	Blackboard	Data
Target component	Blackboard	Source	Computation
Source component	Blackboard	Control	Control
Target component	Blackboard	Source	Computation
Source component	Blackboard	Blackboard	Data
Target component	Blackboard	Blackboard	Control

Now, the *mappings* from Layers to Blackboard have been defined to make transformation possible.

3.3.4 Defining Rules by Q-RDL

The same *mappings* can easily be described by using Q-RDL. The definition of the transformation is always started by defining the “<<NEW TRANSFORMATION>>” tag. Next, the source and target patterns are defined. As the transformation we want to define is Layers to Blackboard, we set two tags, “<<Source pattern>>” and “<<Target pattern>>” and define 'layers' for the source and “blackboard” for the target. The rule list should now look like the Figure 8 presents.

```
<<NEW TRANSFORMATION>>
<<Source pattern>>
layers
<<Target pattern>>
blackboard
```

Figure 8. Defining transformation with Q-RDL – Header.

Next, the *mark mappings* are defined. First, the tags, “<<Source information>>” and “<<Target information>>” are set and then filled with correct marks. In addition to the *marks* presented in Table 5, three more *mark-mapping* pairs are

defined. This is as in Layers; there are two kinds of components with different roles (layer and component) and in this case, the role of the component in Layers does not affect the transformation, thus we define *mark-mappings* for both "layer" and "component" components. Figure 9 presents *mark-mapping* pairs.

<<Source information>> layers component data <<Target information>> blackboard blackboard data	<<Source information>> layers component control <<Target information>> blackboard control control	<<Source information>> layers component computation <<Target information>> blackboard source computation
<<Source information>> layers layer control <<Target information>> blackboard control control	<<Source information>> layers layer data <<Target information>> blackboard blackboard data	<<Source information>> layers layer computation <<Target information>> blackboard source computation

Figure 9. Defining transformation wih Q-RDL – Mark mappings.

The list of the crucial components is defined (see Table 6). First, the "<<Crucial components>>" tag is added and after that an "<<Element>>" tag to separate the crucial elements from each other. The m*arks* of the each crucial element are added then (Figure 10).

<<Crucial components>>
<<Element>>
blackboard
blackboard
data
<<Element>>
blackboard
control
control
<<Element>>
blackboard
source
computation

Figure 10. Defining transformation with Q-RDL – Crucial components.

Connection rules are added as follows: (1) define "<<Connection rule>>" tag, (2) define "<<Source>>" and "<<Target>>" pairs with required information (see Table 11). Defining transformation rules is finished by adding the "<<END TRANSFORMATION>>" tag. Figure 11 presents the connection rules.

<<Connection rules>> <<Source>> blackboard blackboard data <<Target>> blackboard source computation	<<Source>> blackboard control control <<Target>> blackboard source computation	<<Source>> blackboard blackboard <<Target>> blackboard control control <<END TRANSFORMATION>>

Figure 11. Defining transformation with Q-RDL – Connection rules.

Now, the Layers-to-Blackboard transformation is successfully defined by Q-RDL. As it can be seen, defining a new transformation with Q-RDL is easy indeed, as the language reminds how the transformation *mappings* are constructed. In addition, Q-RDL promotes using the quality-driven model transformation technique to define new transformations correctly, as the required data for each transformation follows the same order as the transformation to be defined.

3.3.5 Performing Layers-to-Blackboard Transformation

Before performing a transformation, the following pre-conditions have to be met:

- The model has to be *marked*.
- Both the source and the target patterns are to be defined in the stylebase.
- The transformation is admissible.
- The transformation rules are defined in the rulebase.

After securing pre-conditions, transformation can take place by following the rules defined in the rulebase.

As illustrated by an example, the pre-conditions are met. In this example, the rules were defined with Q-RDL for the transformation (see Figures 8, 9, 10 and 11).

1. Transformation from Layers to Blackboard begins by identifying the correct rule in the rulebase. This is done by matching both the source and the target pattern names with the corresponding fields (Figure 8) in the transformation rules. When the correct rule is found, the actual transformation process can begin.
2. The *marks* attached to the model are changed to corresponding *marks* of the target pattern. The *mark mappings* (see Figure 9) define the source components by defining all the *marks* attached to them and the corresponding *marks* of the target pattern. The m*arks* of the found components are transformed.
3. After *mark mapping*, existence of crucial components (see Figure 10) of the Blackboard pattern is observed. This is conducted by browsing

the model for crucial components. If some component is missing, it is generated. In the example, all the crucial components are found.

4. As the connector topology relies only of the target pattern, all existing connectors are removed. The new connector topology is constructed by following the connection rules (see Figure 11). The connection rules define which kind of component is to be connected to which type of component. A data component is connected to computation components, a controller component is connected to computation components and to a data component.

After constructing a connector topology, the result of the transformation should look after re-arranging the components to correspond to the new architectural pattern, as Figure 2 presents.

4. Evaluation of UML tools for model-driven architecture

In this section, several commercial and open-source tools, which are listed in bullets below, are studied to find the most suitable one to be extended to support the quality-driven model transformation. The tools are selected for the study by listing the best known Computer-Aided Software Engineering tools and by selecting some of the less known ones relatively randomly. The tools have to support Unified Modelling Language and therefore other tools which support some different modelling language are not considered.

At the time of writing this section (February 2004), there are some previous studies on different CASE tools capabilities, but no research has been conducted, or at least not published, from the perspective of supporting MDA and structural modelling. Just some lists of tool support for different features and diagrams exist, as Mr. Mario Jeckle presents in his web site [26]. The following modelling tools are evaluated:

- ArcStyler 4.0
- ArgoUML (2/04)
- iUMLite 2.2
- Jvision 2.1
- Poseidon Pro 2.1.2
- Prosa UML 2004 Programmer edition
- ProxyDesigner 1.0
- Rhapsody Developer 5.0
- Rose Technical Developer
- Rose XDE Developer Plus (2/04)
- Telelogic Tau/Developer 2.2.51
- Together ControlCenter (2/04)
- UMLet 3 beta.

The evaluation is two-phased. First, tools are studied from vendors' website, which is practically the only source of information regarding tools' capabilities. Because of this, every fact, which will be presented leans merely on the vendors' datasheets and sales talks: all information gathered may not be completely

accurate. The most unsuitable tools are then filtered out and the remaining tools are chosen for extension trials.

In the second evaluation phase, the evaluation versions of the remaining tools are obtained. The second phase of evaluation consisted of two tasks: performing all including modelling tutorials and evaluating the tools' extension capabilities. Finally, the tools are compared against each other.

This section is structured as follows: First, the first evaluation framework is presented against which the tools are compared to resolve a couple of the most promising tools. Second, the second evaluation framework is introduced in order to resolve the most suitable tool to be a platform for the tool extension. Finally, the summary of the evaluation is presented.

4.1 The First Tool Evaluation – Literature Study

Table 8 presents the evaluation framework of the first evaluation phase. In architectural modelling, expressing the internal structure of the classes and defining component interfaces are considered essential. For that reason, the UML versions of the tools are observed. As UML 2.0 is currently in the finalization phase and some uncertainty exists on how tool vendors implement it at a moment, the structural modelling capability is observed separately, as well.

As it is intended to implement a tool extension, modelling tools has to provide some kind of extension interface. What kind of interface and what languages could be used for extensions are not considered as essential aspects in the first framework and for that reason these are not observed. Just whether the tools are extendable or not are considered.

As it is defined in MDA, models are ultimately transformed into code. For that reason, code generation for at least one language has to be provided. As code generation is not a straightforward task, there are two kinds of generators: full source code and code template generating generators. The full source code generators should generate all or nearly all of the code, while the code template producing generators only generates class and function templates. Code generation for C, C++ and Java are observed, as it is assumed that these are the most common languages to be used. Only full code generators are considered.

Some features that could lighten and quicken the development, maintenance and testing are also chosen into the evaluation framework. Automatic document generator, which should produce readable documents from designed models, is observed. Support for some kind of testing and debugging environment is observed, too.

Table 8. The first evaluation framework.

Aspect	Question
UML	What is vendor's announced UML version?
StructM	Does the tool support structural modelling?
Ex	Does the tool provide extensibility interface for user-defined plug-ins?
C	Does the tool support code generation for C?
C++	Does the tool support code generation for C++?
Java	Does the tool support code generation for Java?
Doc	Does the tool provide any automatic document generator?
Sim	Does the tool support any testing and debugging environment?

From the perspective of the first evaluation framework, every evaluated tool is reported in Table 9. Mark “X” means “yes” answer to the question presented in the first evaluation framework.

Table 9. Summary of the results of the first evaluation.

Tool	UML	Ex	StructM	C	C++	Java	Doc	Sim
ArcStyler 4.0	1.4	X		X		X	X	
ArgoUML (2/04)	1.3	X						
iUMLite 2.2	1.4	X					X	X
JVision 2.1	1.3						X	
Poseidon Pro 2.1.2	1.4	X					X	
Prosa UML 2004 Prog.	1.5			X	X	X	X	X
ProxyDesigner 1.0	N/A							
Rhapsody Developer 5.0	2.0	X	X	X	X		X	X
Rose Technical Developer	1.4	X	X	X	X	X	X	X
Rose XDE Developer Plus (2/04)	1.4	X					X	X
Telelogic Tau/Developer 2.2.51	2.0	X	X	X				X
Together ControlCenter (2/04)	1.4	X		X	X	X	X	
UMLet 3 beta	N/A	X						

UML is on the threshold of a new era, as version 2.0 is being published in the near future. For that reason, it would not be reasonable to consider further any tools that do not support UML 2.0. On the other hand, Rose Technical Developer does support ports, structure modelling and other typical features of UML 2.0 even with version 1.4. Due to this, it must be considered that a supported UML version does not explicitly reveal the real structural modelling capabilities and for that reason, the supported UML version cannot be used for evaluation.

The most effective way to drop out unsuitable tools is by checking whether the tools support structural modelling. This criterion filters ten tools out from further consideration and leaves Rhapsody Developer, Rose Technical Developer and Telelogic Tau/Developer for further consideration.

The remaining tools do provide some kind of extension interface and support for full code generation for at least one language. Therefore, no tools are dropped out at this stage. Neither are any of the tools filtered out when the support for document generation and for testing and debugging environment were observed as these were considered minor aspects. Due to this, the remaining three tools are selected for a more detailed evaluation.

4.2 The Second Tool Evaluation – Empirical Study

As the first evaluation is based on vendors' datasheets and sales talk only, a few the same features listed in the first evaluation framework are selected to the second evaluation framework to ensure the correctness of information. Again, the purpose is to describe significant characteristics of the tool from the perspective of MDA and extendability, not to show every feature. The second evaluation framework is presented in Table 10.

The UML version and especially the structural modelling capability are taken into the framework to make certain that the tools do support structural modelling, as it is told. This is because vendors often promise more than they deliver.

Extensibility is divided into two separate categories: extensibility interfaces and UML profile extensions. The extensibility interfaces are for accessing the tool through provided application programming interface (API) with some programming language. The UML profile extension is for modifying and extending UML itself.

Platform independent and platform specific modelling is observed when the tool's support for MDA is considered. At this time, the platform considered can be any 3GL or 4GL, so being platform independent, the tool has to provide its own action language for describing the model's behaviour. In this way, a model can be compiled to any supported target languages. If no action language is defined, it is considered that the tool only allows platform specific modelling.

Table 10. The second evaluation framework.

Aspect	**Question**
UML	What is vendor's announced UML version?
StructM	Does the tool support structural modelling?
ExtL	What languages can be used for tool extension?
Profiles	Does the tool provide support for defining new UML profiles?
MDA	In what extent does the tool supports MDA?

Three tools – Rhapsody Developer, Rose Technical Developer and Telelogic Tau/Developer – are selected for closer evaluation. Telelogic Tau/Developer and Rhapsody Developer are evaluation versions, downloadable completely free from the vendors' web site, whereas Rose Technical Developer is a commercial version.

The evaluation is performed as follows. First, the tools are installed and after that, the tools are evaluated one at the time. The tool evaluation consists of two tasks: First, all the included modelling tutorials are performed. Second, the tool extension tutorials and some modifications of our own are done to get better acquainted with the extension interfaces.

From the perspective of the second evaluation framework, the evaluated tools are reported in Table 11.

Table 11. Summary of the results of the second evaluation.

Tool	UML	StructM	ExtL	Profiles	MDA
Rhapsody Developer	2.0	Only C++	COM, VBA	X	PSM
Rose Technical Developer	1.4	X	OLE, RRRTS		PSM
Telelogic Tau/Developer	2.0	X	COM, TCL	X	PIM, PSM

The vendors' announced UML version does not seem to be an important aspect when ranking, as the tools do support ports and structural modelling whether the version is 1.4 or 2.0. Currently, Rhapsody Developer only supports structural modelling when working with C++ language.

The tools support at least two extension mechanisms for plug-ins, therefore the count cannot be used for ranking. Nor can the number of extension languages be used, as there are plenty of where to choose from in all cases. API cannot be assessed either, as only trivial extensions were made during the tool evaluation. Because of that, there is no experience implementing full-fledged plug-ins and for that reason, some uncertainty remains on every provided API. According to the documents, the tools are freely extendable, so they have to be considered as equals at this stage. Instead, support for creating new UML profiles can be used for consideration.

Tau/Developer is the only tool which supports platform independent development, as no target code has to be written anywhere. This is due to the fact that Tau/Developer provides its own platform independent action language. Whereas in Rhapsody Developer and Rose Technical Developer, the behaviour of state machines and classes' operations has to be implemented with the target language. In Tau/Developer, the platform specific issues have to be taken care of just when the model is integrated with the actual environment.

When MDA is considered, none of these tools support it in all its forms. Although Tau/Developer allows platform independent developing, it does not support platform specific modelling as it is defined in MDA. There is no transformation from PIM to PSM defined in any way. PIM can be made into a PSM by writing an inline target code into the model, but no transformation takes place, or at least no automatic transformation. In fact, the model is more like a

blend of PIM and PSM than just plain PSM, as platform independent action language still exists in the model. On the other hand, if the source code is considered a platform specific model, then a clear transformation from PIM to PSM exists. A direct transformation from PIM to code is also defined in MDA, so no standard is violated. However, the whole developing cycle from CIM to PIM and from there to PSM and finally to the code does not exist.

The two remaining tools do not support MDA in any of its forms, as at least one PIM has to exist when Model-Driven Architecture is considered. On the other hand, if the platform is defined as an operating system, then these two tools do support MDA. In summary, the support for MDA is just a matter of definitions.

Overall, Tau/Developer is considered the most suitable one, as it is the only one, which allows platform independent developing. Rhapsody Developer and Rose Technical Developer are quite similar tools, as neither of them can be used for platform independent developing. Some differences occur in structural modelling, as Rose Technical Developer has its capsules whereas Rhapsody Developer structure is designed straight into classes. There is actually no specific reason why one should be preferred above the other, but we lean towards Rhapsody Developer, as it supports creating new UML profiles. For that reason, Rhapsody Developer is ranked the second and Rose Technical Developer the third.

4.3 Summary

Thirteen CASE tools were studied to find the most suitable one to be extended to support quality-driven model transformation. The tools had to support UML 2.0 or at least structure modelling. In addition, an extensibility interface was required. These two criteria filtered ten unsuitable tools out and left three for further evaluation. Telelogic Tau/Developer, Rhapsody Developer and Rose Technical Developer were evaluated one at the time and later compared against each other.

Tau/Developer allows platform independent developing by including its own action language to describe the model’s behaviour completely, but also target code can be written in any place if desired. Tau provides two extension

interfaces for one's own plug-ins and allows defining new UML profiles. Thus, the extension possibilities are relatively unlimited.

Rhapsody Developer and Rational Rose RealTime are in the most part quite similar. Platform independent developing is not possible as no strong action language is included and the target code has to be written in to describe behaviour. There are also restrictions in Rhapsody Developer’s modelling capabilities, as it only supports class structure modelling when working with C++. Both tools support two extension interfaces for plug-ins, but only Rhapsody Developer allows defining new UML profiles.

Tau/Developer seemed to be the most suitable tool; Rhapsody Developer was considered the second and Rational Rose RealTime the last. None of these tools is incompetent, but the main reason why Telelogic Tau/Developer achieved the first place is that it makes platform independent developing possible, whereas it is not possible with the other two.

5. Development of the Q-Tra tool

The aim of applying the quality-driven model transformation technique is to enable automation of the transformation process [5]. Without tool support, automation is not possible. This section introduces a tool extension, Quality-driven architecture TRAnsformation tool (Q-Tra), to Telelogic Tau/Developer.

The technique describes transformation between two platform independent models. The source model where from the transformation is to be taken is a model, which is designed certain quality-attributes in mind and implemented with carefully considered design solutions – design and architectural patterns. All elements participating in a certain pattern in the source model are marked with the required *marks* for making the application of the technique feasible. The purpose of the Q-Tra is to help the architect in choosing a new architecture for a system by offering a set of alternative solutions, which promotes certain quality attributes. In addition to guiding the architect in making wise decisions, the Q-Tra provides a possibility of performing the desired transformation.

The Q-Tra is discussed as follows: First, the requirements for the Q-Tra are presented. Requirements are concerned with what the Q-Tra is supposed to do, and what is required of the operating environment. In addition, some requirements for the implementation are presented. Second, the design of the tool extension is discussed. The design of the Q-Tra is concerned with how the tool relates to its environment, what kind of architecture it has, components roles and their interoperability. Finally, the solution is presented by introducing implementation and testing of the components.

5.1 Requirements for the Tool Extension

The requirements of the tool extension can be divided into three categories: (1) end-user requirements, (2) requirements for the modelling tool and (3) other technical requirements. The end-user requirements are concerned with what an architect or developer needs. Requirements of the modelling tool are concerned with the special requirements the tool extension sets for the modelling tool. Finally, the technical requirements of the tool extension are related to the

implementation of the end-user requirements and the architecture of the tool extension.

5.1.1 End-User Requirements

The implementation of the quality-driven model transformation clearly relies on a few aspects, which have to be implemented in the first version of the tool extension. The first version of the Q-Tra tool is a prototype, no requirements are set for performance nor any other special requirements for usability etc. Based of that scoping, the following essential requirements for enabling model transformation were defined:

1. UML model has to be able to browse for different kinds of entities and diagrams. It must be possible to find all the design and architectural patterns, which are applied in the model. In addition, an end-user should be able to select, which attributes constrain the search. At the end, the user has to be able to see the search result.
2. To store all the design and architecture patterns, some kind of data storage has to exist. In addition to being a completely passive pattern repository, the user may want to add new patterns, remove and edit the existing ones. For every pattern, the following data has to be stored: pattern name, reference, quality attributes, component types, component roles, connector types, data topology, control topology, purpose, diagram name, abstraction level and rationale.
3. The user wants to perform a quality-driven model transformation between two patterns. Conducting a transformation between user-chosen patterns should require as little as possible user attention. However, semi-automatic transformations are allowed, as the nature of the quality-driven model transformation may restrict the making of completely automatic transformations. No demands on component topology after the transformation are set, as re-arranging components to conform to the new architecture may not give good results. That is, it is assumed that an architect wants the components to stand at the same place after the transformation. In this way, the architect does not need to re-locate the components. In addition, it may be easier to

observe the changes in the model if the component topology is not modified.

5.1.2 Modelling Tool Requirements

In addition to the general requirements presented in Section 4 for the modelling tool, a few specific requirements are set because of the quality-driven model transformation technique:

1. The modelling tool must allow adding information, *marks*, to the components in the model. In addition to *marking* the model, the tool API must have access to the *marks*. No restrictions for the place, where the *marks* are located, are set.

2. Due to the quality-driven model transformation, support for the following UML 2.0 diagram types is essential: deployment, composite structure, class and state machine. These are the diagram types where the transformation will take place. In addition, either the modelling tool or the end-user must validate the syntax and semantics of the model, as ill-formed models may cause peculiar errors at some stage.

5.1.3 Technical Requirements

The tool extension has to be implemented in a modular way because there is uncertainty:

- what modelling tool will be used as the basis of the tool extension
- what kind of user interface should be implemented
- how the stylebase and other possible databases are implemented
- the whole transformation technique itself.

Thus, all components should be designed in a way that replacing one component does not affect to the other ones and if it does, the impact should be minimal. In addition, the modelling tool, where the tool extension is used, should be

replaceable. Thus, promoting loose coupling between components is essential when designing the architecture for the tool.

The user interface of the tool extension should be graphical. It must be possible to conduct the following tasks:

- adding, removing and editing patterns in the stylebase
- browsing the stylebase
- browsing patterns on the basis of the quality attributes they promote
- selecting which pattern to be searched from the model
- selecting which pattern to be transformed.

The following requirements are defined for the stylebase:

- All patterns have to be defined in the same stylebase.
- Implementation technology is not limited.

The technical requirements of the tool extension were defined abstractly, because the tool extension was developed parallel to the development of the quality-driven model transformation technique. Practically, the uncertainty of the final implementation of the components of the tool extension and the capability of implementing the transformations at issue, can be considered the driving forces of the Q-Tra.

5.2 Design of the Q-Tra Tool Extension

This section describes the technology dependent constraints and architectural solutions of the Q-Tra tool. The purpose is to introduce what components the Q-Tra consists of, what their responsibilities are, and how they interact.

5.2.1 Technical Constraints for Designing the Q-Tra

Programming Constraints

Telelogic Tau/Developer's COM interface provides full access to its UML model. COM is "a platform-independent, distributed, object-oriented system for creating binary software components that can interact". [27] Thus, as an implementation language, any given COM enabled language, such as Visual Basic and C++, can be used for writing tool extensions. We chose C++ although there was no extensive experience. In addition, there was no previous experience on COM interfaces. However, the choice was clear as Telelogic Tau/Developer only provides a minimal set of tool extension examples and all the provided examples are implemented by C++. Documentation of extension interfaces is minimal and some of the examples are even erroneous. Considering previous facts, the only rational choice is to implement the tool extension with the above-mentioned technology. Telelogic Tau/Developer also provides TCL API, but it was not even considered, as it is meant for simpler scripting extensions.

Tool API

Telelogic Tau/Developer can be considered a meta-model driven tool, i.e. the structure of the tool repository is based on the publicly available metamodel. A metamodel is a set of metaclasses and meta-attributes that defines the conceptual view of the information stored in the model. For instance, the model consists of classes and their interaction. In a metamodel, the classes are described explicitly in a way that they can be used in the model. That is, a metamodel presents the vocabulary of the language and its usage. This also reflects the way in which the tool repository is accessed.

COM API only provides a small set of general-purpose primitives for accessing the elements in the model. The elements in the model are formed of other elements by inheriting them. In addition, all the elements have just a small set of basic primitives that can be accessed. For example, to get all comments attached to a specific entity, we can write:

comments = entity->GetEntity("ModelElement")->GetEntities("Comment")

Here, “entity” means, for example a class in the class diagram, of which we want to extract comments. As the entity presents just a class symbol, and no actual class model element, we apply the GetEntity(“ModelElement”) method in order to get the class’s parent class, which contains a comment field. When we have the ModelElement, we can apply the GetEntities(“Comment”) method in order to access one of its meta-attributes called “Comment”. The comment consists of a set of comments – entities. In order to access the first comment of the comment list we must apply two more methods:

comment = comments->GetItem(1)->GetValue(“Text”)

The GetItem(1) method is applied in order to access the first line, the first entity, of the comment list. Then, we apply the GetValue(“Text”) method to the comment entity to get the actual comment string, which was the first comment line of the entity we accessed.

Here, the “Text” meta-attribute is the only primitive in the whole sequence that can be considered a variable in the general sense. All the other model aspects in Tau are called Entity classes or Entities, which present a collection of Entity classes. Thus, all the methods for all the elements in the model are the same. In this way, the API becomes simple, but knowing the structure of the UML 2.0 metamodel becomes essential. In fact, there is roughly a fistful of methods that are most generally used in accessing the model but there are tens of metaclasses and meta-attributes in the metamodel that have to be known.

Integration of a Tool Extension

All COM plug-ins are introduced to Telelogic Tau/Developer in the following way:

- A special add-on introduction file has to be written to inform Tau/Developer about the new plug-in. The main purpose of this introduction file is to describe the location of the TCL script, which is used to start the actual tool extension. The TCL script describes the place to which in the menu structure the start menu item of the add-in is to be attached. In addition, it describes the actual tool extension, which is to be launched when the user clicks the menu item. TCL

script is also responsible for initiating the tool extension by sending the necessary parameters to it. However, the contents of these files are outside the scope of this thesis.

- When designing an interactive tool extension into Tau/Developer, the client plug-in has to implement a certain interface, which is accessed when the tool extension is started. When starting the tool extension, Tau/Developer gives two parameters. The first one is a pointer to an application which works as a server for the client. The second parameter is a pointer to the actual UML model. This means that a UML model is not accessed by sending a request through API to Tau/Developer, but manipulating the model directly inside the tool extension with the provided API. This also means that the model is only sent to the tool extension once at the beginning. This affects the implementation of the tool extension.

Quality-Driven Model Transformation Technique

The technique assumes that components of the platform independent model have to be *marked* in order to apply the technique for the transformation. As stated, *marks* include three aspects: the name of the pattern where the component participates in, component role and type. As Tau/Developer provides a way for extending and refining UML with profiles, the most logical choice for the *marks* would be creating a new UML profile for quality-driven model transformation technique, which would add new fields for *marks* to the components of the model. However, the *marks* were added to the comment field of the components, as the method how to access the new fields added by a new UML profile remained unknown.

5.2.2 Architecture of the Q-Tra

Designing architecture for the Q-Tra begins from identifying the necessary components to realize the quality-driven model transformation. An example transformation presented in Section 3.3 suggests the use of two databases: one for storing all the style and pattern information and one for storing the rules. The interfaces for both of these databases have to be implemented. In addition, an interface to Tau/Developer has to exist. For human interaction, a graphical user

interface (GUI) has to be implemented. A component responsible for controlling all other components has to be implemented between the interface layer and GUI. Considering the constraints, the conceptual architecture of the Q-Tra can be constructed (Figure 12 [5]).

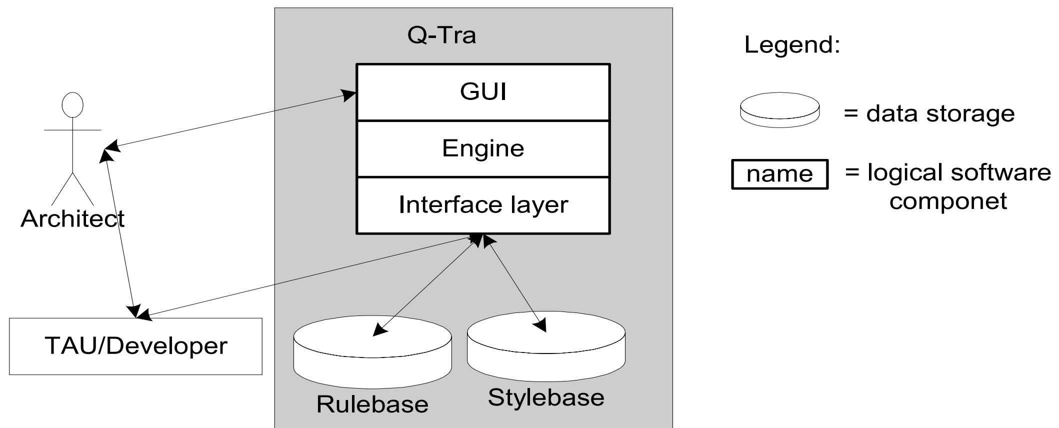


Figure 12. Conceptual architecture of the Q-Tra and its context.

The Q-Tra consists of three layers (Figure 12): GUI, Engine and Interface layer. The Interface layer provides access to both databases and to Tau/Developer. An architect using the Q-Tra through the user interface has also access to Tau/Developer for modelling purposes. The Engine component between GUI and the Interface layer coordinates and controls transactions between GUI, Tau/Developer and databases.

The technical requirements for the Q-Tra architecture demanded that the architecture of the tool should be designed to promote loose coupling between components. In addition, it should be possible to replace the components with a minimal impact on the implementation of the others. Thus, all tasks have to be implemented with separate components.

The graphical user interface of the Q-Tra can be considered a component but the Interface layer has to be divided into three separate components. As there are two kinds of databases – stylebase and rulebase – there will also be two database access handlers. It can be concluded that only one component needs access to the rulebase. Thus, we define a separate component, which is responsible for

both handling the access to the rulebase and conducting the transformations. In addition, accessing the UML model is divided into its own separate component. As all the tasks are divided into their own separate components, the only role of the Engine component is to connect all the components together and to work more like a router between components.

Tau/Developer requires a certain kind of interface to be implemented in order to start the Q-Tra and to send the necessary parameters; a separate component for this task is defined. The responsibility of this component, in addition to providing interface to Tau/Developer, is to encapsulate all the modelling tool specific data in a way that no other component than this one and the component responsible for accessing Tau/Developer needs to be changed, if the modelling tool is changed.

Overall, the Q-Tra has six components, which play different roles in the architecture. Table 12 summarises the responsibilities of the components and estimated changes that have to be conducted if something varies on either the environment or the other components.

Table 12. Summary of the components.

Name	Responsibility	Interdependency
UIHandler	Provides graphical user interface	Does not affect the other components
Engine	Router	Interfaces
DatabaseHandler	Stylebase access handler	Does not affect the other components
ModelHandler	Accesses the UML model	Complete reconstruction of ModelHandler if the modelling tool changes
Transformer	Rulebase access handler, conducts transformations	Does not affect the other components
CTtdAddIn	Tau connection point, encapsulates the modelling tool specific parameters	Causes complete reconstruction of ModelHandler if the modelling tool changes

Architecture of the Q-Tra has to be designed to provide loose coupling between components by hiding their implementation from each other. This is achieved by applying a well-known behavioural pattern called *mediator* [12]. The *Mediator* pattern states that only two components know each other and all components interacts among themselves through one central component. By applying the *mediator*, the changed implementation of one component is not shown in the other components. Moreover, there is no need to change the components' responsibilities that have already been decided. Therefore, the *mediator* pattern seems more than suitable for the architecture of the Q-Tra.

By applying the *mediator*, the component topology takes the following form (Figure 13): At the centre, there is the Engine, which works as a router or *mediator* for the whole tool extension. All the other components are connected to the Engine component only. There are no connectors between components.

Figure 13 also describes four connection points to the outside world. DatabaseHandler communicates with the environment, in this case with the stylebase, through the StylebaseAccess port. Transformer has its own communication channel to the rulebase. The communication port of the UIHandler describes the communication with the end-user. The communication direction is only to inside the Q-Tra, as the purpose is to present the end-user driven interaction. That is, all the graphical data is presented by UIHandler but the interaction from the end-user comes from the outside world. This is the reason why there is only a provided interface. The fourth environment interface is from CTtdAddIn. This connection point describes the communication with Tau/Developer. In practise, through this port Tau/Developer makes its first access to the Q-Tra and sends the compulsory parameters. As there is no outgoing message exchange between the Q-Tra and Tau/Developer in general, there is no required inteface.

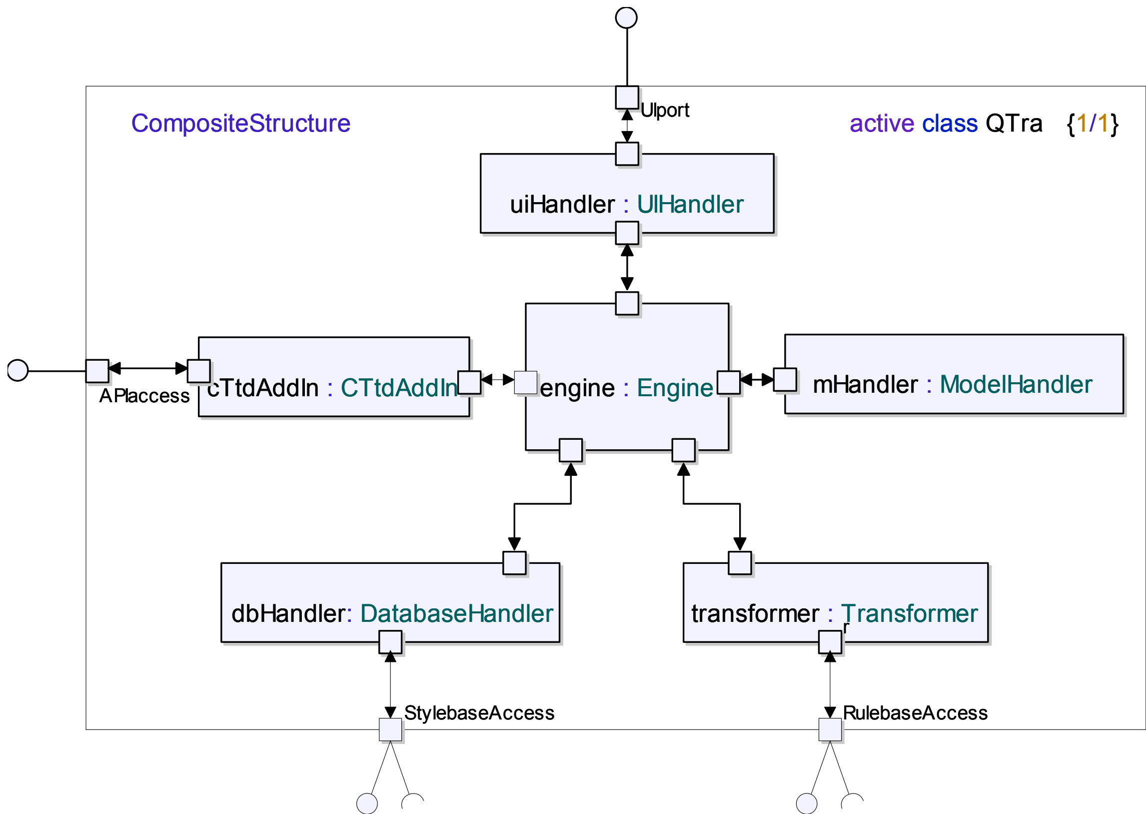


Figure 13. Architecture of the Q-Tra.

Encapsulating all specific parameters of the modelling tool into one specific container object promotes modelling tool independency. That is, when Tau/Developer first starts Q-Tra, it sends two parameters to the CTtdAddIn component. CTtdAddIn encapsulates those parameters into one container object and sends it to the ModelHandler, which is the only component in addition to CTtdAddIn that is aware of the modelling tool. In this way, replacing the modelling tool should affect the other components as little as possible, if the connection to other modelling tools is implemented in somewhat same way as it is with Tau/Developer.

Figure 14 presents, how the Q-Tra is conceptually started and created. First, Tau/Developer sends ITtdInteractiveServerPtr and ITtdEntitiesPtr to CTtdAddin (signals from env[1] to cTtdAddIn[1]) in order to start the Q-Tra. ITtdInteractiveServerPtr is a pointer to the application, which works as a server

for the Q-Tra and ITtdEntitiesPtr is a pointer to the particular UML model. Then, CTtdAddIn encapsulates these parameters into an object called ToolSpecificParameters, creates Engine and sends ToolSpecificParameters to it. This is the only task that is set for CTtdAddIn. After that, Engine creates all other components and sends ToolSpecificParameters to ModelHandler.

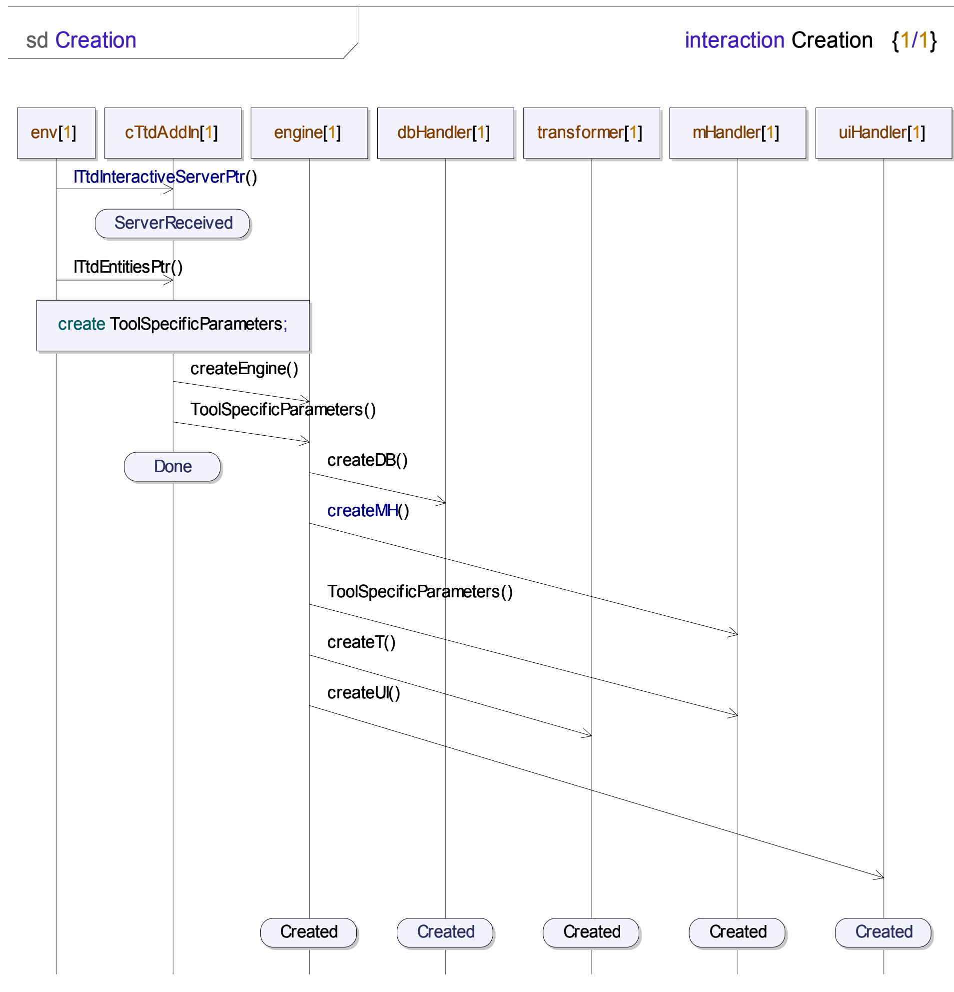


Figure 14. Initiation of the Q-Tra.

All requirements for the architecture of the Q-Tra are met. However, the overall idea of the signal propagation in the Q-Tra needs to be elucidated. Clarification of the roles of the components is necessary before continuing further. Therefore, a quite extensive example of the workflow is presented.

Model transformation is the most laborious task of all the use cases. As stated in Section 3.1, conducting transformation composes of the following tasks: applying admissibility rules, finding the right rule for transformation and conducting the transformation at issue.

From the point of the Q-Tra, admissibility rules are applied by fetching both the source and the target patterns for transformation from the stylebase and then by checking if the patterns share the same abstraction level and purpose. If applying the admissibility rules results positive, the transformation can be performed. This is done by fetching the correct rule from the rulebase and by conducting the transformation by following the transformation rules.

Figure 15 presents the message propagation of the Q-Tra when the transformation is conducted:

1. An architect selects the transformation between certain patterns from the user interface.
2. UIHandler forwards the transformation request to Engine, which routes the transformation request to Transformer.
3. In order to validate the transformation admissibility, Transformer requests the source and the target patterns from Engine. The request is forwarded to DatabaseHandler.
4. DatabaseHandler performs a query to the stylebase for the required patterns and returns them back to Engine, which forwards them to Transformer.
5. Transformer validates the transformation admissibility by applying the admissibility rules to the patterns. In this case, the transformation is admissible.
6. In order to conduct the transformation, transformation rules are fetched from the rulebase. The transformation rules contain all the necessary information for conducting the transformation.
7. The final task is to guide ModelHandler in accessing and modifying the UML model. This is performed by sending a series of guidance signals to ModelHandler via Engine.

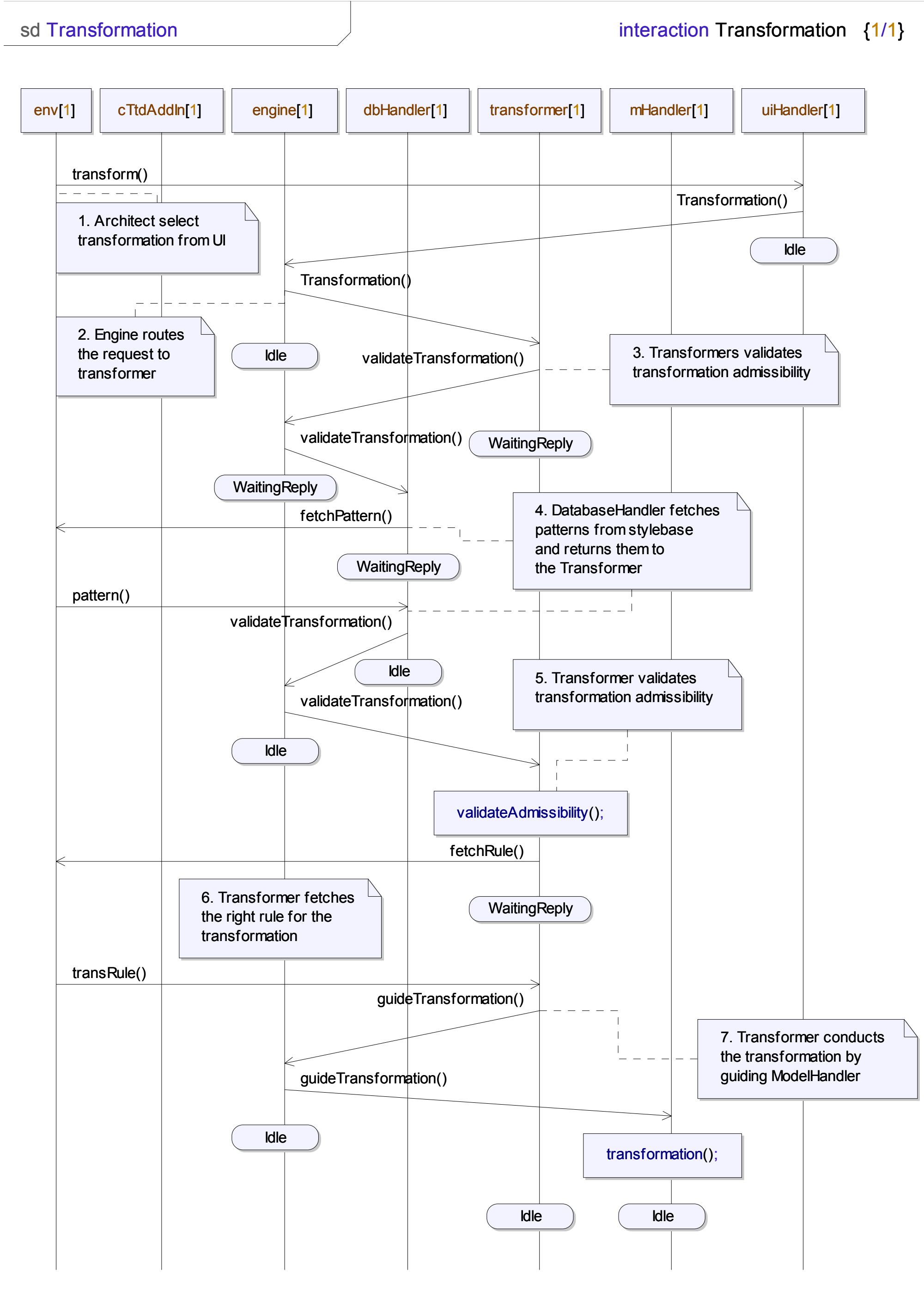


Figure 15. Transformation workflow of the Q-Tra.

When examining the roles and responsibilities of the components and interactions of the components, the most important concrete signals and interfaces can be defined. The interfaces and contents of the signals are described in the following order: The interface and signals to DatabaseHandler are described. Then interfaces and signals to Transformer and ModelHandler are described. CTtdAddIn and UIHandler are discussed next. Interfaces of Engine implements all interfaces of the other components, as all the signals go through it.

DatabaseHandler

DatabaseHandler is responsible for accessing the stylebase and replying to queries. Since the implementation of the stylebase can vary, only the provided interface to the Engine is defined and the definition of the interface to the stylebase is left to the implementation stage. DatabaseHandler provides basic services for accessing the stylebase:

1. Add new element.
2. Remove element.
3. Query elements.
4. Load stylebase.
5. Save stylebase.
6. Clear stylebase.

All services are to be implemented with their own functions, thus the interface consists of six different functions. Services 1, 2 and 3 takes parameters called Element when applied. Element is a special object, which contains the fields of one element in the stylebase, information about one design or architectural pattern. Service 1 applies Element as such and adds a new pattern into the stylebase. Service 2 queries the stylebase with the information gathered in Element and deletes the desired pattern. Service 3 queries the stylebase with the information gathered in Element and returns all the patterns, which match the provided information. The query result is returned as a special object called ResultSet, which consists of series of Elements. Services 4 and 5 take the parameter character string used in loading and storing the stylebase. The character string could be, for example, a filename from which the stylebase is loaded or to which the stylebase is stored, or IP address where the stylebase is

located. The last service is used when the entire stylebase must be cleared. Table 13 summarises the interface of DatabaseHandler.

Table 13. Summary of the interface of DatabaseHandler.

Service	Name	Input	Output	Responsibility
1.	setElement	Element	Boolean	Adds new element into stylebase
2.	deleteElement	Element	Boolean	Removes element from stylebase
3.	query	Element	ResultSet	Queries elements from stylebase and returns query result
4.	loadDatabase	Character string	Boolean	Loads stylebase
5.	saveDatabase	Character string	Boolean	Stores stylebase
6.	clear		-	Clear entire stylebase

Transformer

Transformer is responsible for accessing the rulebase and guiding ModelHandler in performing transformations. In the case of the stylebase, implementation of the rulebase can also vary freely; the required interface to the rulebase is not defined. However, the provided interface to Engine is defined. Transformer is an independent controlling component and thus its interface remains simple. Transformer provides two services:

1. Check transformation admissibility.
2. Conduct transformation.

Service 1 takes two character strings as parameters. The first parameter defines the source pattern name and the second the target pattern name. Transformer performs two queries to the stylebase in order to get both the source and the target patterns. Transformer applies the transformation admissibility rules according to information and validates the transformation admissibility. Service 2 conducts the transformation according to the parameters it gets, when applied. Three parameters are given: two character strings for source and target pattern names for the transformation, and a list of entities, which are participating in the

transformation. The list of entities consists of a group of objects called Entity. Entity consists of *marks* and information, which can be used to locate the parts explicitly in the UML model. Table 14 summarises the interface of Transformer.

Table 14. Summary of the interface of Transformer.

Service	Name	Input	Output	Responsibility
1.	checkPatternCompatibility	2 Character string	Boolean	Validates transformation admissibility
2.	transform	2 Character string, entityList	Boolean	Conducts transformation

ModelHandler

The responsibility of ModelHandler is to provide access to the UML model. ModelHandler is the only component which is allowed to handle a model, and all queries and model handling related functionality are conducted here. As the way the model is passed – at least with Tau/Developer – and handled in ModelHandler, there is no outgoing interface to the environment of Q-Tra. Thus, the model is accessed *inside* ModelHandler. However, the interface between ModelHandler and Engine is defined explicitly with all tasks needed to conduct a transformation. The following services must be provided:

1. Initiate modelling tool specific aspects.
2. Fetch entities from the model.
3. Modify marks attached to the entities.
4. Create new entity into the model.
5. Clear obsolete connectors between entities in the model.
6. Connect entities in the model.

ModelHandler is initiated by service 1. The service takes one parameter, ToolSpecificParameters, in order to get knowledge of the UML model and all other modelling tool related aspects.

Service 2 has two character strings as parameters. The first parameter is for identifying and fetching entities participating in a certain pattern in the model.

The second parameter defines the abstraction level of the entities. This is for identifying a diagram which the query is supposed to apply. The result of the query is an Entity list.

Service 3 attaches *marks* to a specific entity. The entity is selected by giving one Entity parameter when the service is called. Entity describes explicitly the entity to which the *marks* are attached. *Marks* are given in the second parameter.

Service 4 takes one Entity parameter and creates a new entity into the desired location in the model.

Service 5 uses the Entity parameter to specify the diagram name from which all connectors of the entities participating in a certain pattern are removed.

Service 6 specifies two entities that are to be connected together with a connector. Information for this task is given by two Entity parameters. Table 15 summarises the interface of ModelHandler.

Table 15. Summary of the interface of ModelHandler.

Service	Name	Input	Output	Responsibility
1.	setParameters	ToolSpecific-Parametes	-	Initiates modelling tool specific parameters
2.	fetchEntities	2 Character strings	Entity list	Fetches entities from model
3.	setPatternParameters	Entity, Character string array	Boolean	Modifies marks attached to the model
4.	createEntity	Entity	Boolean	Creates new entity to the model
5.	clearObsolete-Connectors	Entity	Boolean	Clears connectors between entities
6.	connectComponents	2 Entity	Boolean	Connects entities

CTtdAddIn

The CTtdAddIn component is a compulsory modelling tool specific component, and its interface to outside the Q-Tra is always defined by the modelling tool, to which

the Q-Tra is integrated. However, interfaces between Engine and CTtdAddIn should be the same. CTtdAddIn provides interface to the modelling tool and a packaging and forwarding service for the modelling tool specific parameters. The interface for sending ToolSpecificParameters is on the required interface of CTtdAddIn. The Interface for CTtdAddIn consists of the following services:

1. Provide an interface for the modelling tool.
2. Forward modelling tool specific parameters.

When Tau/Developer is the modelling tool to which the Q-Tra is integrated, two signals are sent to the Q-Tra at the beginning. Service 1 provides an interface to the modelling tool. Service 2 is used to forward ToolSpecificParameters onwards. Table 16 summarises the interfaces of CTtdAddIn.

Table 16. Summary of the interface of CTtdAddIn.

Service	Name	Input	Output	Responsibility
1.	raw_OnExecute	ITtdInteractiveServerPtr, ITtdEntitiesPtr	-	Interface
2.	setupModel-Handler	-	ToolSpecific-Parameters	Tool specific parameters

UIHandler

As the nature of the user interface is to master and control the other components by sending series of messages and requests, UIHandler does not provide any services for the other components inside the Q-Tra. However, the required interface is described explicitly, i.e. interface provided by Engine.

The purpose of the user interfaces is to the control other components. The services required by UIHandler reflect the interfaces of the other components inside the Q-Tra. Thus, the tasks needed to perform transformations and handle databases are included in the required interface of UIHandler. The following services are required by UIHandler:

1. Add new stylebase element.
2. Remove stylebase element.

3. Query stylebase elements.
4. Load stylebase.
5. Save stylebase.
6. Clear stylebase.
7. Fetch entities from model.
8. Check transformation admissibility.
9. Perform transformation.

Rationales for services 1 to 6 are the same as the interface of DatabaseHandler. The service 7 stands the same as service 2 in the interface of ModelHandler, and for services 8 and 9 the same interface rationale stands as provided by Transformer. Table 17 summarises the required interface of UIHandler.

Table 17. Summary of the interface of UIHandler.

Service	Name	Input	Output	Responsibility
1.	dbSetElement	Boolean	Element	Adds new element into the stylebase
2.	dbDeleteElement	Boolean	Element	Removes element from the stylebase
3.	dbQuery	ResultSet	Element	Queries elements from the stylebase and returns query result
4.	dbLoadDatabase	Boolean	Character string	Loads the stylebase
5.	dbSaveDatabase	Boolean	Character string	Saves the stylebase
6.	dbClear			Clears entire stylebase
7.	mFetchEntities	Entity list	2 Character strings	Fetches entities from the model
8.	tCheckPattern-Compatibility	Boolean	2 Character string	Checks transformation admissibility
9.	tTransform	Boolean	2 Character string, entityList	Conducts transformation

Engine

The Engine component acts as a junction point for all the other components and all signals travel through it. For this reason, Engine implements all the interfaces

of the other components, i.e. Engine uses the services of the other components and every component can access the services of the rest of the components through Engine. Figure 16 summarizes all the interfaces of Engine.

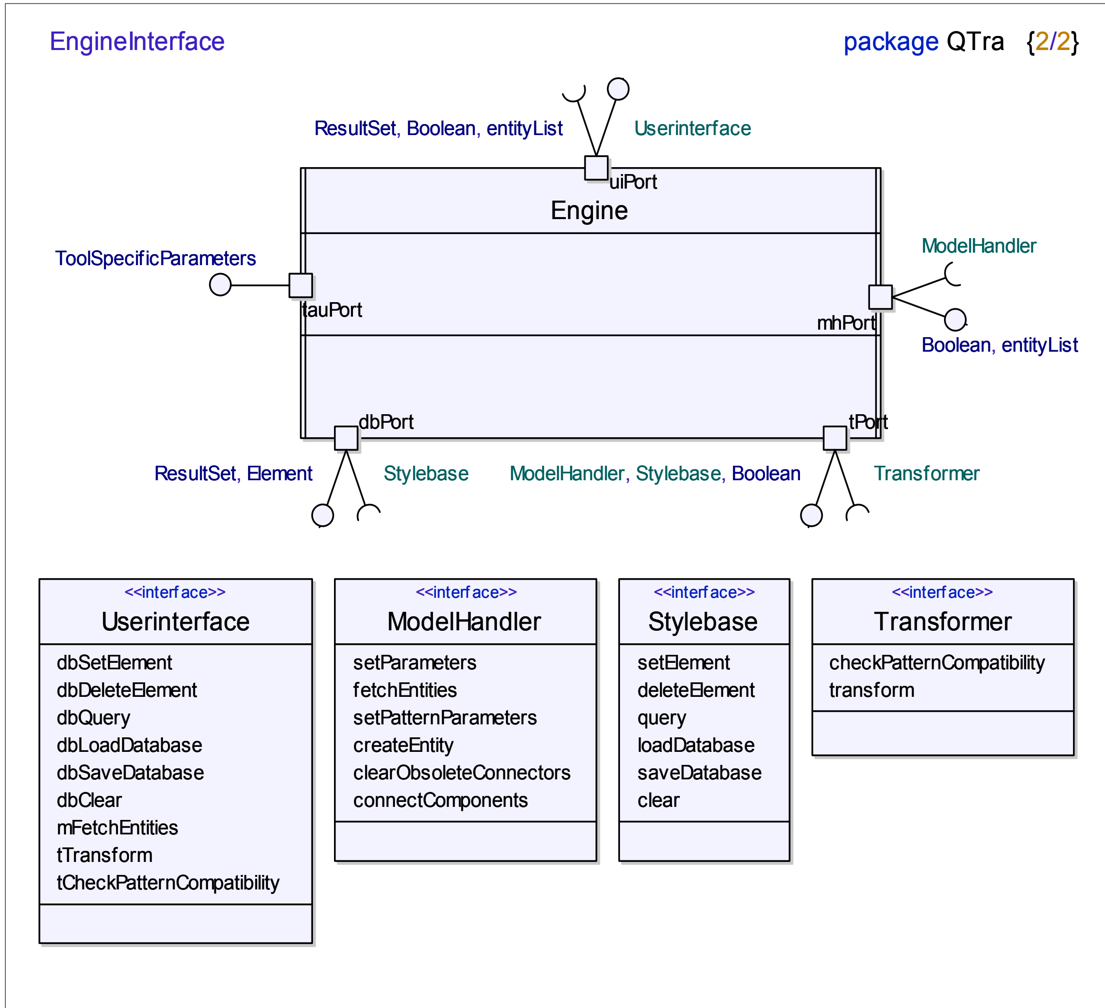


Figure 16. Interfaces of the Engine.

In Figure 16, both the provided and required interfaces of the components can be seen. Although all the interfaces of the components are discussed above, the interfaces of Transformer requires clarifying.

Transformer is a control component, whose nature is to control other components, while the rest of components are passive, except UIHandler. By observing tPort in Figure 16 (bottom right of Engine), it can be seen that the required interface implements Transformer interface, which consists of the interface declared above, but the provided interface implements two other interfaces. When Transformer applies admissibility rules [5], it fetches both the source and the target patterns from the stylebase. For this reason, there is also a Stylebase interface. Transformer guides ModelHandler during transformations, and thus there is a ModelHandler interface.

5.3 Implementation of the Q-Tra Tool Extension

As stated in Section 5.2.1, C++ was selected for the implementation language. For the development environment, Visual Studio 6.0 was chosen, as it provides a good environment for writing plain source code and creating graphical user interfaces. Next, the implementation of the components is discussed. The purpose is not to explicate the source code of classes and components. The purpose is to introduce how the design plan is realized.

5.3.1 Implementation of the Components

DatabaseHandler

The implementation of DatabaseHandler begins from examining possible database solutions for the stylebase. At the beginning, Structured Query Language (SQL) [28] based database solutions seemed to be the best choice, as there was some previous experience implementing databases with the current technology. SQL was abandon for the following reasons: the Q-Tra was to be designed and implemented at the same time as the quality-driven model transformation technique was developed. Thus, the implementation process would definitely go through some iterations and later lead to some changes in the stylebase structure. Therefore, the stylebase should be implemented in a way that it could be easily modified. The easiest way to realize a highly modifiable stylebase is to design and implement it by yourself, i.e. by constructing the database from the scratch.

As there were no special requirements for the implementation of the stylebase, such as distribution, performance and high-end database, a linked list based solutions seemed to be sufficient. A Linked list can be described as a list of objects connected linearly together in order to form a dynamically changeable list structure. For nodes in the list, there would be classes called Element. As stated, (see Section 5.2.2) Element contains the fields required for specifying one design or architectural pattern. The idea is to keep the list in the main memory at the run time and to store it to the disk when necessary. For the data saving format, a text file was chosen to promote modifiability. Appendix 3 presents the contents of the tag-based solution of the stylebase.

By choosing a trivial solution for the stylebase, implementing DatabaseHandler becomes easy. There is no need for setting up database servers or any such thing; just maintaining a linked list is sufficient.

Transformer

The implementation of Transformer is highly dependent on how the rules for the transformations are described. In this case, the rules are described with Q-RDL and thus Transformer remains simple. This is because the rules defined by Q-RDL contains all the information necessary for conducting transformations and thus no complement semantics for guiding the transformation has to be coded to Transformer.

The rulebase is implemented in the same way as the stylebase i.e. as a linked list based object database, where the transformation rules are nodes of the list. The rules are described by the objects called TransRule, which contain the aspects defined in one transformation rule. The rulebase is accessed in the same way as the stylebase.

Transformer, being a component responsible for accessing and handling the rulebase, does not provide a service for adding, removing or updating rules such as DatabaseHandler does. This is because there is still an enormous amount of uncertainty how the transformation rules will be described and Transformer may have to be re-designed and re-implemented. Therefore, Transformer is left to remain as simple as possible. The Transformer can only read the rulebase, and

the new transformation rules have to be written by using some third-party text editor.

Transformer is also responsible for applying the admissibility rules and conducting transformations. Transformer controls the other components and thus the implementation consists of function calls to other components through Engine.

Transformation process consists of the following tasks:

1. Fetch the rcorrect transformation rule from the rulebase.
2. Change marks of the model.
3. Generate missing entities to the model.
4. Remove connectors between entities that are to be re-connected.
5. Create new connector topology.

ModelHandler

When the Q-Tra is started, ModelHandler receives the ToolSpecificParameters object from Engine. ITtdInteractiveServerPtr is a pointer to the COM servers, in this case to Tau/Developer and ITtdEntitiesPtr is a pointer to the entities of the UML model.

In order to get to the root of the UML model, the following code is written:

root = entities->GetItem(1)->GetEntity("Session")->GetEntity("root");

After getting the root entity, it is possible to start browsing the model. Currently, a composite structure diagram is the only diagram accessed.

The implementation of ModelHandler consists, for the most part, of model browsing, i.e. fetching entities from the model and fetching parameters from the entities. In order to manipulate the elements in the model, the following steps are performed:

- Fetching the entities from the model begins from the root entity.

- Next, the diagrams of the model are browsed in order to find the correct diagram.
- When the correct diagram is found, the entities in it are browsed.
- After the correct entity is found from the diagram, the right attribute (or entity, for instance port is an entity of a part) is located.
- When the correct attribute is found, it can be manipulated.

As stated, ModelHandler is the only component allowed to access the UML model and its implementation is strongly driven by the modelling tool, to which the Q-Tra is attached. Understanding the implementation of the functions and services of ModelHandler requires knowledge of COM API and UML 2.0 metamodel, therefore no implementation details are presented here.

CTtdAddIn

Telelogic Tau/Developer dictates a lot of the implementation of the CTtdAddIn. The purpose of CTtdAddIn is to

- provide an interface to Tau/Developer
- package the modelling tool specific parameters
- create Engine component in order to start the Q-Tra.

Providing the interface for Tau/Developer is performed by implementing a certain interface called ITtdInteractiveClient. In addition to implementing an interface to Tau/Developer, CTtdAddIn implements COM specific interfaces. However, these are not discussed here, because COM specific issues such as the interface and objects are outside the scope of this thesis. More information on COM can be found in [27].

Encapsulating all the modelling tool specific parameters, which are ITtdInteractiveServerPtr and ITtdEntitiesPtr, is realized by creating a special container object ToolSpecificParameters, which contains space for the parameters. In this way, ToolSpecificParameters acts like a shuttle for the parameters, and no other components, except for ModelHandler, needs to know anything of its contents.

The last task of the CTtdAddIn is to create the Engine component in order to start and initialize the rest of the components of the Q-Tra. This is conducted by reserving memory for it and by sending ToolSpecificParameters. The Engine component then commands and no other tasks are performed in the CTtdAddIn.

UIHandler

Tau/Developer allows adding self-made plug-in applications directly into its menu structure. Thus, the start button of the Q-Tra is added there.

The Q-Tra tool extension has a dialog based graphical user interface, i.e. the Q-Tra is a pop-up program. There are two kinds of dialogs. The modal dialog is a window, which retains the focus until it is explicitly closed. The modeless dialog is a window, which does not require closing before switching to another window.

The modeless dialog would be a better choice for the GUI, as it allows switching between the Q-Tra and Tau/Developer on the fly, and it would not be necessary to close the Q-Tra while doing other tasks with Tau/Developer, but there were some problems in implementing the GUI with it. Tau/Developer became extremely unstable and the reasons for the peculiar behaviour were never reasoned out. Therefore, the modal dialog type was chosen. However, starting the Q-Tra is easy; the modal dialog is not a bad choice.

Implementation of the GUI consists of the main dialog (Figure 17), which is used to start other task specific dialogs. The task specific dialogs are concerned with accessing the stylebase and conducting the transformations.

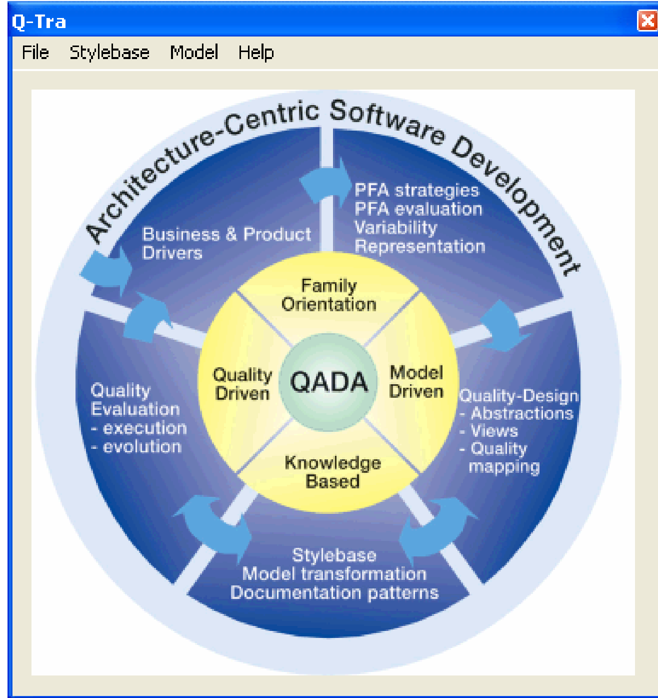


Figure 17. Main dialog of the Q-Tra.

Adding, removing and updating contents of the stylebase are performed by using separate dialogs. Figure 18 presents the dialog, which is used for adding new design and architectural patterns into the stylebase. The dialog contains the information necessary for describing patterns. Four buttons at the bottom of the dialog are used to browse the stylebase and to add a new pattern into it. The dialog is responsible for validating that the necessary fields are filled in order to add a new pattern into the stylebase. If some fields are left blank, the dialog requests the architect to fill in the blank ones. Remove and update dialogs function similarly as the add dialog.

Figure 18. Add dialog.

Transformation is performed using the transformation dialog (Figure 19). The transformation dialog consists of three sections and two buttons:

- Source pattern information field
- Target pattern information field
- Found component window, which shows components participating in the source pattern

- Fetch button, which is used to perform queries to the UML model
- Transform button, which is used to conduct transformation.

Source and target pattern sections consist of three fields:

- Name of the pattern
- Quality attributes
- Abstraction level switch, which defines the abstraction level of the patterns shown in the pattern name list.

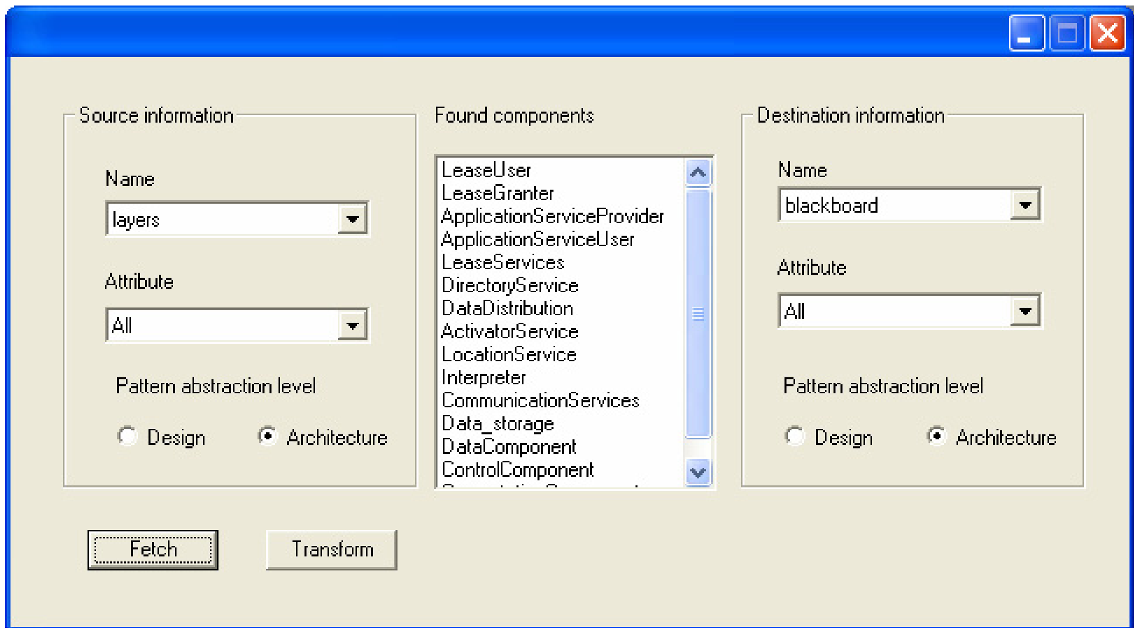


Figure 19. Transformation dialog.

Querying the stylebase for patterns and quality attributes is easy as the pattern name field and the quality attribute field are updated dynamically: The architect can select "All" from the quality attribute field and browse the pattern name field in order to show all design or architectural patterns the stylebase contains. On the other hand, if the architect selects "All" from the pattern name field and browses the quality attribute field, it shows all quality attributes that are found in the stylebase. Moreover, if the architect selects, for instance, "modifiability" from the quality attribute field, the pattern name field is updated with a list of

patterns, which promotes this quality attribute. Again, by selecting a certain pattern from the pattern name field, the quality attribute field is updated with all the attributes the pattern promotes.

Updating dynamically the pattern name field and the quality attribute field results in good usability after some practise. For instance, if the architect wants to find all the patterns, which promote "portability", he or she selects "All" from the pattern name field and "portability" from the quality attribute field. Now, the architect can browse the pattern name field for all patterns that promote the desired quality attribute. Selecting one pattern, the architect can browse what other quality attributes the pattern promotes and make the final decision by considering all the quality attributes.

The architect can perform a query to the UML model by (1) selecting one pattern in the source pattern field and (2) pushing the "Fetch" button. The Q-Tra browses the model and returns list of components participating in the pattern. The architect can double click the components in the list in order to see some more information about them (Figure 20). The transformation is conducted by pressing "Transform" button.

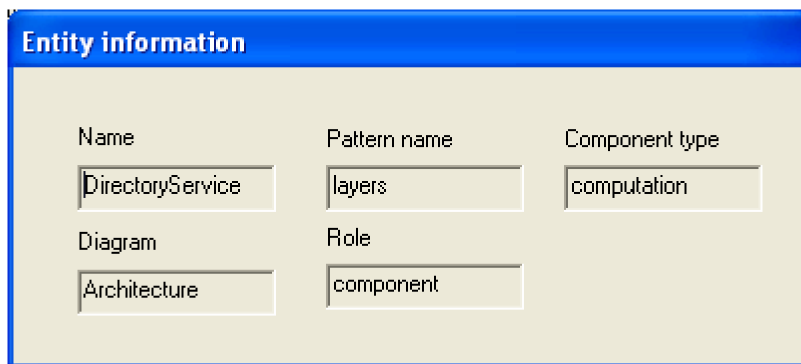


Figure 20. Entity information field.

Implementation structure for the user interface consists of some task specific classes, which are activated by a class representing the main dialog. Communication to Engine is implemented through an interface, which is common for all the user interface classes.

Engine

Engine provides a loose coupling between components by routing a service to another. In addition, Engine is responsible for creating and deleting the other components in the Q-Tra. In practice, the Engine creates a component and gives its pointer to the component in order to make bi-directional communication possible. Thus, the created component has access to Engine at the same way as Engine has access to the created component. Engine also implements the interfaces of the other components. Thus, the created components see Engine with the services of the other components. In this way, interdependency of the components is minimized.

5.3.2 Testing the Components

The testing of the Q-Tra implementation was carried out in many different phases during the development of the tool extension. Testing the functionality and correctness of the components was performed with black-box approach, i.e. sending series of inputs to the component and observing the results it returned. Once the tests were successfully passed and the faults were corrected, the work continued with implementing the next component.

The first component that was implemented and tested was DatabaseHandler, which is responsible of accessing the stylebase. The tests covered the basic services of simple database, i.e. loading and saving elements, performing queries and adding new elements to the data repository.

The second component that was implemented and tested was the CTtdAddIn, which provides an interface to Tau/Developer. The tests to the component remained trivial, as CTtdAddIn does not contain any complex behaviour. After this, ModelHandler was implemented and tested. As the ModelHandler is responsible for accessing and manipulating the UML model, extensive tests to this component were difficult to perform. This is because there could be numerous different kinds of anomalies in UML models that the architect constructs if correctness of the model is not verified. Therefore, the tests performed at this point covered only correctly build models and handling various anomalies were left for further development.

The fourth component that was implemented and tested was UIHandler, which provides a graphical user interface. As UIHandler accesses the other components, the mediator component, Engine, was also created at this point. Now, the first integration tests were carried out. The last component that was implemented and tested was Transformer. The tests remained more or less the same as with DatabaseHandler. Implementing all components successfully resulted in the final testing of the Q-Tra functionality.

6. Case study – layers-to-blackboard transformation

For illustrating quality-driven model transformation with the Q-Tra, a simple case study is presented. The case is called Distribution Service Platform (DiSeP). The purpose of the DiSeP is to make the software components in a networked environment to interact spontaneously. Components in the DiSeP are various kinds of services that are either a part of the platform or a part of the application that utilizes the platform. The configuration of the network may change dynamically. That is, the number of modules or the range of the available services may change. The main goal of the DiSeP is to maintain the interoperability of the services despite the dynamic nature of the network. [13] Figure 21 presents the conceptual architecture of the DiSeP.

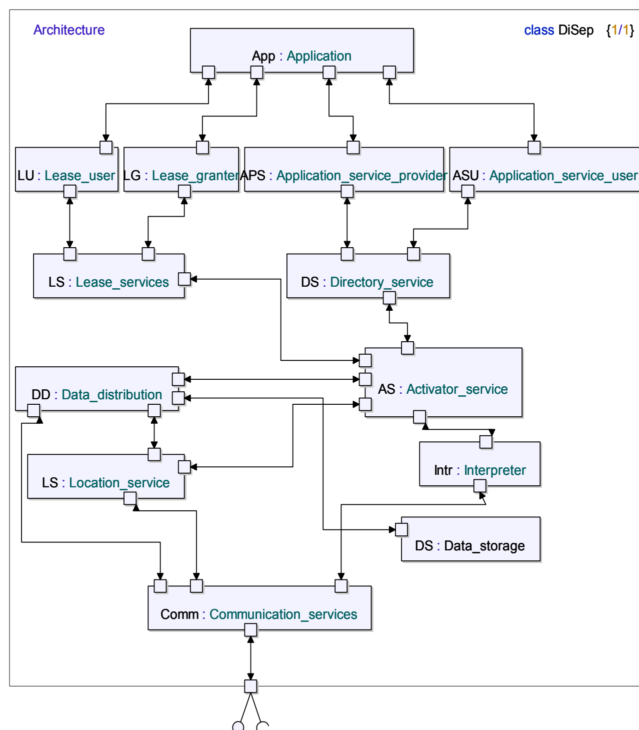


Figure 21. DiSeP – Source model.

At the top of the Figure 21, stands Application, which illustrates the application using the DiSeP. The layer below Application contains four interface

components, which provide interfaces for services that can be directly accessed by the application. Furthermore, the layer below the interface layer contains two components. The Lease service utilizes the lease management and the Directory service provides a directory for distributed data storage. The most complex layer contains five components, which are responsible for receiving and processing the incoming control information and sending the outgoing control information. The Activator service monitors the state of the network, the Data storage works as distributed data storage, the Interpreter encodes and decodes XML messages, the Data distribution operates the data storage and the Location service manages the location information specific aspects. The last layer, the Communication service provides services that handle communication between different units in the network.

The architecture of the DiSeP applies the Layers architectural pattern. In order to apply quality-driven model transformation, the source model has to be *marked*. By knowing the roles and types of the components, the *marks* of the source model can be defined. The *Marks* of each component (Table 18) are added to the comment fields of the components in the model.

Table 18. Related component marks in the source model.

Component	Style	Role	Type
Application	-	-	-
Activator service	Layers	Component	Control
Application service provider	Layers	Component	Interface
Application service user	Layers	Component	Interface
Communication services	Layers	Layer	Computation
Data distribution	Layers	Component	Computation
Data storage	Layers	Component	Data
Directory service	Layers	Component	Computation
Interpreter	Layers	Component	Computation
Lease grantor	Layers	Component	Interface
Lease user	Layers	Component	Interface
Lease service	Layers	Component	Computation
Location service	Layers	Component	Computation

The Q-Tra provides means for easy resolving of an optimal architecture for the system. This is done by following the six steps defined below:

- Open transformation dialog from the Q-Tra main window.
- Press “Design” or “Architecture” pattern abstraction level switch in both the source and the target information fields in order to begin browsing patterns in the desired abstraction level.
- Select the current architecture from the source pattern name field and set “All” to the source attribute field.
- Select the desired quality attribute from the target attribute field in order to update the target pattern name field with all patterns that promote the current attribute.
- Browse the target pattern name field for the target architecture candidates and select one.
- Validate the other quality attributes of the target architecture pattern candidate by browsing the target attribute field.

After selecting the target architecture for the system, the transformation can be conducted. The transformation is performed by the following two steps:

- Press “Fetch” button in order to query the model for components participating in the source pattern.
- Press “Transform” button to transform the source model to the target model with the desired architecture.

In this example, we present how to change quality attributes of the architectural model of DiSeP to promote extensibility. As stated in Section 3.3.1, the Layers architectural pattern promotes modifiability, portability and reusability. Thus, Layers pattern is not an optimal solution for the architecture of DiSeP if extensibility is considered essential.

In order to resolve optimal architecture for DiSeP, we follow the steps defined above. First, (1) the transformation dialog is opened (see Figure 19) from the Q-Tra. As we want to manipulate the architectural specific aspect of the DiSeP, we (2) press “Architecture” of the pattern abstraction level switches in both the source and the target information fields. For (3) the source pattern name field “layers” is selected, as the DiSeP currently utilizes the Layers pattern. For

attribute field, “All” is chosen. In order to find the architectural solution, which utilizes extensible architecture, (4) “extensibility” is selected form the target attribute field. By selecting the desired attribute from the target attribute field, browsing target pattern name field for pattern candidates is possible. Browsing the target pattern name field reveals that the Blackboard pattern promotes the desired quality attribute. By (5) selecting “blackboard” from the target pattern name field, the target attribute field is updated with the quality attributes that the Blackboard architectural pattern promotes. The target attribute field (6) reveals that, in addition to extensibility, Blackboard promotes availability, modifiability, reliability and reusability, thus Blackboard seems to be a good choice for the new architecture of the DiSeP.

In order to start the transformation process, we (1) fetch all entities, which participate in the source, Layers, architecture of the DiSeP by pressing the fetch button. The result is a list of components (see Figure 19) participating in the current pattern. Here, we can see the components which will take part in the transformation process. Now, everything is set for transformation.

Transformation is performed by (2) pressing the transform button. The Q-Tra takes control, makes its computations and carries out the transformation from Layers to Blackboard pattern. Figure 22 presents the result of model transformation.

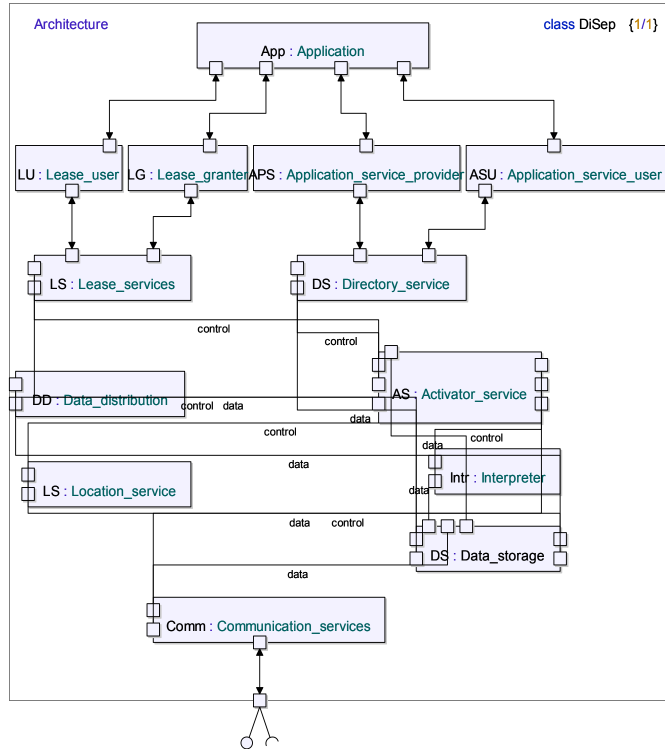


Figure 22. DiSeP – Target model after transformation.

In model transformation, the *marks* of the components are also transformed to correspond the new purposes of the components. Table 19 presents *marks* that are attached to the target model.

Table 19. Related component marks in the target model.

Component	**Style**	**Role**	**Type**
Application	-	-	-
Activator service	Blackboard	Control	Control
Application service provider	Layers	Component	Interface
Application service user	Layers	Component	Interface
Communication services	Blackboard	Source	Computation
Data distribution	Blackboard	Source	Computation
Data storage	Blackboard	Blackboard	Data
Directory service	Blackboard	Source	Computation
Interpreter	Blackboard	Source	Computation
Lease grantor	Layers	Component	Interface
Lease user	Layers	Component	Interface
Lease service	Blackboard	Source	Computation
Location service	Blackboard	Source	Computation

As it can be seen in Figure 22, components in the model are not relocated; just the connector topology is modified in addition to the *marks* (see Table 19). By inspecting the *marks* of the components, it can be noticed that not all components are transformed, as there still are interface components with pattern name mark “Layers” attached. The reason is that no rule was found for transforming interface components and thus they are left out of the process.

Since the component topology is not rebuilt in the transformation and only the connector topology is considered in the graphical presentation, some of the architect’s attention is required to make the model more expressive. After re-arranging the components and adjusting the component topology, the model takes a new form, which reminds the Blackboard architectural pattern (Figure 23).

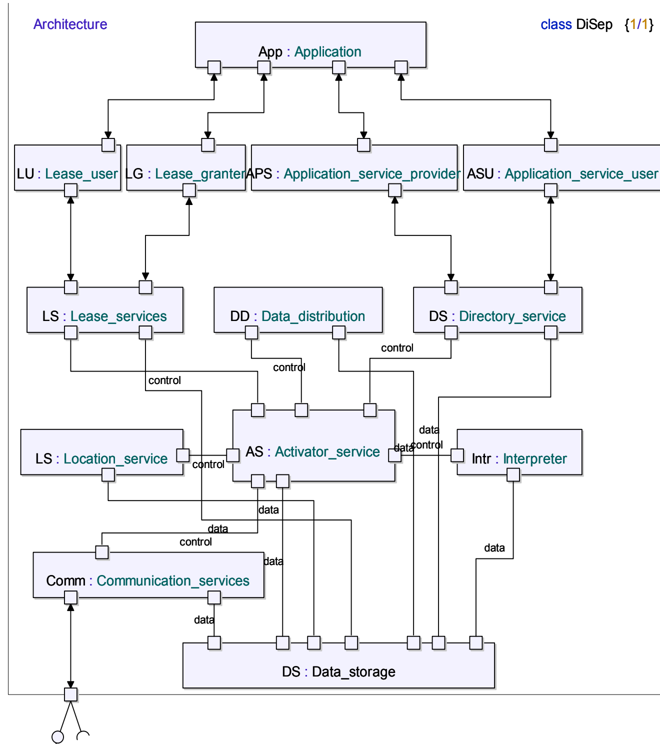


Figure 23. DiSep – Target model after rearranging components.

In Figure 23, it is easy to discern the Blackboard architectural pattern. At the centre of the model the controller component, Activator service, can be seen to control the other components. The Data storage in the role of data component can be found at the bottom of the diagram.

7. Discussion

The quality-driven model transformation is one attempt to bridge the gap between the quality properties and the architectural structures. As no transformation technique is effective without a proper tool support, the goal is to automate the transformation with advanced CASE tools.

In this thesis, we presented the Q-Tra tool that automates the quality-driven model transformation. Although the Q-Tra and the transformation technique is far from ready, the time for the first trials on automating quality-driven model transformations is here. Currently, we have managed to implement one horizontal transformation from the Layers architectural pattern to the Blackboard pattern successfully, but in the near future, more transformations will be defined. In addition to defining more architectural transformations, the aim is to adapt the quality-driven model transformation technique and its automation for transforming the design patterns. The most ambitious goal of the horizontal transformation would be to extend the technique for the model dynamics, i.e. transforming in addition to the structure of the system, the behaviour.

In this thesis, we have shown that the transformation technique and the quality-driven model transformation are realizable. This is a tiny step bridging the gap between quality attributes and architectural structures but far from insignificant. Significance of the work comes from breaking the ice. If a complete quality-driven model transformation, i.e. transforming the model structure and the behaviour could be realized, the benefits would be revolutionary. It would be possible to optimize whole systems for the desired quality properties just by a press of a button.

Although we have managed to realize the quality-driven model transformation, the time for further development is not favourable due to a change of generation in the modelling languages. At the time the modelling tool evaluation was performed (see Section 4.), UML 2.0 was in the finalization phase. Currently (December 2004), the situation has not changed, as UML 1.5 is still the official version. In addition to the confusing state of the modelling language standard, modelling tool vendors implement the upcoming UML 2.0 and its new features differently and with a certain delay. On the top of that, new features often come up immature: such is the case with Tau/Developer. Whether the transformations

are vertical or horizontal, a lot of work is still required in order to accomplish the goals of the MDA. Considering the facts above, development of the automated quality-driven model transformation is on the crest of the research area.

Next, we analyse the work done against the research problems that were presented in Section 1. In addition, future development of the Q-Tra tool will be discussed.

7.1 Experiences in Applying Quality-driven Rule Description Language

Several restrictions were encountered in the automation while applying the Q-RDL for transformation description. As a case, we look at the Blackboard-to-Layers transformation.

Mark mappings between patterns work correctly, as they are one-to-one *mappings*. The problems arise while trying to construct a new connector topology, as in Layers, components do not have pre-defined types, roles or are in any pre-defined order. It cannot be defined that a certain data component must have access to a certain computation component. We can define that all data components have access to computation components, but not to a certain computation component or vice versa.

This is surely a restriction of the Q-RDL, but a completely automated transformation to Layers from any arbitrary pattern can be questioned. This is because the transformation to Layers is always a one-to-many or even many-to-many transformation and these kinds of transformations may not be even possible without user interaction. Currently, expressing the need of user interaction in a certain phase of the transformation with the Q-RDL can be easily done by just writing the desired field, for instance, “user interaction required”.

Currently, the Q-RDL lacks support for expressing all kinds of anomalies, except for missing crucial components. For instance, in the Layers to Blackboard transformation if there are two data components instead of one in Layers, what to do with the other one, as in Blackboard only one data component, blackboard,

is allowed? How to express it that you have to get rid of the other data component if that is wanted?

Restrictions of the Q-RDL can easily be understood, as there is even no other experience of applying the quality-driven model transformation technique than to defining Layers-to-Blackboard transformation. However, Q-RDL will evolve at the same phase as the technique does, and the restrictions in it will diminish when the quality-driven model transformation techniques mature. Currently, it seems that Q-RDL can be used at least for the one-to-one transformation as such.

As transformations defined by using the Q-RDL result in pair-specific transformation rules, which is also the case when applying the quality-driven model transformation technique for transformation definitions, may lead to problems. Adding one new pattern into the pattern repository, where the transformation between the patterns in both directions (source-to-target, target-to-source) must be defined, may result in a workload explosion. This is because *n* patterns, where *n* is the total number of patterns, have *n!* possible pair combinations and when one new pattern is added into the pattern repository, this multiplies of the number of possible new combinations by *n+1*. For instance, if at the beginning we have three patterns in the pattern repository and they have *n!*, that is six combinations, after introducing one new pattern into the repository there is total of *(n+1)!*, i.e. 24, pair combinations.

Despite the restrictions and possible workload explosion of applying Q-RDL, it is still a viable solution for describing transformation rules defined by applying the quality-driven model transformation technique. However, it may be abandoned when standard transformation description languages emerge.

7.2 Analysis of the Tool Evaluation Result

In Section 4, thirteen UML modelling tools were evaluated to find the most suitable one to be extended with a tool that automates the quality-driven model transformation. The evaluation was two-phased: First, the evaluation iteration was performed in order to get a general view of the available modelling tools

and to filter out the most unsuitable ones. The second iteration was conducted to the remaining tools to resolve the most suitable one to be extended.

The evaluation of the modelling tools emphasised two aspects: extendability and support for MDA. The extension capabilities of the tools were observed by checking whether the tools offer support to API for plug-ins and support for making new UML profiles. Support for MDA was considered by checking whether the tools allow platform independent developing. If the tool provided an action language which does not restrict the ultimate implementation language, it was considered that the tool supports to PIM extend of MDA. Otherwise, the tool only supports the PSM extend of MDA.

Telelogic Tau/Developer was considered the most suitable modelling tool to be extended and the rest of the tools were abandon. In order to analyse the success and accuracy of the evaluation result, some experiences gained of Tau/Developer while designing and implementing the Q-Tra are presented.

7.2.1 Experiences of Using Telelogic Tau/Developer

At the beginning when the first design plans were made, it was thought that the Q-Tra would be implemented by the MDA approach. That is, the source code for the tool extension would have been generated from the model, or at least of a part of it. However, the MDA approach was given up for certain reasons.

When the simplified model of the Q-Tra was realized and simulated (see Figures 13, 14 and 15) with Tau/Developer, it seemed that the model cannot be compiled to source code directly, but the model had to be refined with some extra information. These did not include *marks* (see Section 2.3.4), but some action code had to be written here and there just for the purpose of creation of components and other objects on the model. This was considered peculiar, as the simulation did work well without any special component creation code. In addition, the main method had to be written to make the program start. Neither of these was considered a burden, but some questions occurred about what else would have to be done in order to get the program running. However, these are not flaws of the modelling tool. The uncertainty of everything rose from the lack of expertise. Yet, there are certainly loads of bugs in Tau/Developer.

While writing the main method to the model, some peculiar software bugs were encountered. It seemed that writing certain aspects into the main method caused corruption of the model. This was extremely annoying, as the model could not be resurrected. After consulting the Telelogic support, they admitted that there certainly is a bug in Tau/Developer and it will be corrected in some upcoming version.

It was clear that the Q-Tra could not be completely constructed by modelling, as at least the graphical user interface had to be implemented in a more traditional way. Therefore, importing external code to the model was tried. At this stage, the lack of documents and expertise on using the tool backfired again. Tau/Developer imports external code correctly but it does not complete the importing process, i.e. it does not save the imported library anywhere. By manually saving the imported library to a separate file, importing and compiling the code agreed to work. This was not a bug of the modelling tool. It was more like a thing that has to be known, because there is no reference in Tau/Developer's help that suggests saving the imported libraries before compiling the model. Similar cases were encountered every now and then, as documents were not sufficient or were in some cases even erroneous.

There are also some bugs in the modelling. A model consists of two views: a diagram, which shows a graphical presentation of the model, and a model browser, which shows the model as a tree. If a user removes connectors between ports from composite structure diagram, the connectors are not removed from the model browser. If the user wants to show the connectors again in the diagram view, he/she has to know which part is connected to which part, as an auto layout feature for connectors does not work. This affected the later implementation of the Q-Tra, as the auto layout feature had to be implemented by ourselves. After consulting the Telelogic support, they admitted a bug and promised to correct it.

When considering and summarising the experiences gained so far, it seemed that Tau/Developer is not ready for extensive utilization if there are serious flaws even in the small and simple models. Despite the bugs and lack of documents of Telelogic Tau/Developer, the evaluation of the modelling tool results still hold up. Tau/Developer delivers all features that were promised.

7.3 Future Development of the Q-Tra

The purpose was to automate the quality-driven model transformation in a CASE tool. The experience gained so far of the Q-Tra has shown that the automated quality-driven model transformation is realizable to some extend. However, development work of the Q-Tra is far from ready, as there are clearly some subjects which require further attention. The following aspects are considered:

- Implement the stylebase and the rulebase with distributed databases.
- Refine user interface.
- Replace Telelogic Tau/Developer with another modelling tool.

7.3.1 Databases

Currently, the stylebase and the rulebase are implemented with a linked list based solution. However, it seems that the databases have to be implemented with distributed databases, i.e. the databases would be located in a separate server. This is because, the architects may develop new patterns and transformation rules, which should be shared with the others. Updating the pattern and rule repositories is easier when the data are located in just one place. In addition, at least the stylebase may be used by other tools developed in the future.

Interaction between the database server and the clients, i.e. the architects, can be conducted at least in three ways:

1. Queries to the distributed database are performed every time when needed.
2. Contents of the database are loaded when the Q-Tra tool is started.
3. Contents of the database is only loaded once and saved to the clients' hard drive. The clients' databases are updated in some period.

The first method requires an online connection between the client and the server in order to function. The information exchange should be minimal, as only queried patterns are transferred from the server to the client.

The second method also requires a connection to the server, but after the contents of the database are loaded, the connection is disconnected. In this way, a client has always up-to-date contents of the database. However, loading database every time to the client may take some time. Furthermore, it is considered unnecessary, as it is assumed that the databases will not change often.

The third method does not require online connection, as the database is loaded only once at the first time when the Q-Tra is launched. Connection to the server is required only when an architect wants to update his/her database.

7.3.2 User Interface

The user interface provides a way of modifying contents of the stylebase and of performing queries and transformations. Currently, the user interface is not designed from the point of view of usability; thus, it will be refined.

At this moment, modifying the contents of the stylebase is implemented with three dialogs: add, remove and update dialog. The dialogs will be implemented with a single dialog, which allows all editing tasks.

Now, querying the model gives the results in a list of components participating in a certain pattern. The list will be replaced by a tree view of the components, i.e. nodes will be diagram names and the components participating in the diagrams are leaves. Querying may be conducted by fetching all known patterns from the model instead of fetching the pattern at the time. In this way, an architect can much more easily discern what patterns are used in the diagrams than by clicking every component in the list and after that resolving the use of the patterns.

Currently, the transformation is performed for all components participating in a certain pattern. This should be refined to support selective transformation. That means that from a tree view, an architect can select the diagrams and components, which will take part in the transformation.

7.3.3 Modelling Tool

Telelogic Tau/Developer was extended to support the quality-driven model transformation by introducing the Q-Tra to it. However, Tau/Developer may be replaced in some stage by another modelling tool. As stated, it is assumed that the only changes that have to be performed in the Q-Tra are replacing the CASE tool accessing point (CTtdAddIn) and re-writing ModelHandler, which is responsible of accessing the UML model. Thus, the accuracy of the estimated changes will be tested.

8. Summary

The automated quality-driven model transformation comes with many uses: An architect can easily experiment and try different kinds of architectures for a system while designing a model just by a press of a button, when traditionally changes in the model have to be made manually. Particularly in the context of product families, quality-driven model transformation is justified. This is, as products of a product family may have various customer groups desiring different qualities from a product. The automated quality-driven model transformation enables easy optimization or change of the desired quality property of a product.

The aim of this thesis was to develop a tool that automates the quality-driven model transformation. In order to accomplish this, three actions were carried out:

- to develop a rule description language for describing the rules defined by the quality-driven model transformation technique
- to find the most suitable CASE tool to be extended with a support for quality-driven model transformation
- to design and implement a tool extension, which automates the transformation.

In order to encapsulate rules for the transformation, defined by the quality-driven model transformation technique, a simple rule description language, Q-RDL, was developed. The basic idea of Q-RDL is to define transformations as pattern-pair specific rules. Currently, transformation from Layers architectural pattern to Blackboard has been defined by applying Q-RDL.

For purpose of finding the most suitable modelling tool to be extended, thirteen CASE tools were studied. The tools had to support UML 2.0 or at least the structure modelling. In addition, an extensibility interface was required. These two criteria filtered ten unsuitable tools out and left three for further evaluation. Telelogic Tau/Developer, Rhapsody Developer and Rose Technical Developer were then evaluated one at the time and later on compared against each other. As a result, Telelogic Tau/Developer seemed to be the most suitable one and therefore it was chosen to be extended.

The last phase was to implement a tool extension, the Q-Tra, to Telelogic Tau/Developer. The Q-Tra assumes that the software architecture is available as a *marked* platform independent model in the CASE tool. For every component, the *marks* include the name of the pattern where the component participates in, the role and type of the component. With use of the *marks* in the model, the contents of the stylebase and the the transformation rules described with Q-RDL, the Q-Tra can make transformations. Currently, the transformation from the Layers architectural pattern to the Blackboard pattern has been defined and implemented.

To conclude, we have managed to automate the quality-driven model transformation with the Q-Tra. However, the development work is not ready. Both the Q-RDL and Q-Tra still need a lot of work to make more transformations feasible and user friendlier.

References

[1] Selic, B. The Pragmatics of Model-Driven Development. IEEE Computer Society. IEEE Software, 2003. Pp. 19–25.

[2] Selic, B. Model-Driven Development in the Embedded Environment with OMG Standards. Presentation in the second international summer school on MDA for embedded systems, Brest, Brittany in France, 6th September, 2004.

[3] Miller, J. & Mukerji, J. MDA Guide Version 1.0.1. Object Management Group, 2003. 62 p.

[4] Ramljak, D., Puksec, J., Huljenic, D., Koncar, M. & Simic, D. Building enterprise information system using model driven architecture on J2EE platform. In: Proceedings of the 7th International Conference on Telecommunications, IEEE, 2003. Pp. 521–526.

[5] Matinlassi, M. Quality-driven Architecture Model Transformation for the Software Product Families. Submitted to the Journal of Software and Systems Modelling, 2004. 32 p.

[6] Merilinna, J. & Matinlassi, M. Evaluation of UML Tools for Model-Driven Architecture. In: 11th Nordic Workshop on Programming and Software Development Tools and Techniques, Turku, Finland: Åbo Akademi, 2004. Pp. 155–163.

[7] Bass, L., Clements, P. & Kazman, R. Software Architecture in Practice. Reading, Massachusetts: Addison-Wesley, 1998. 452 p.

[8] Bosch, J. Design and use of software architectures: adopting and evolving a product-line approach. Harlow: Addison-Wesley, 2000. 354 p.

[9] van der Linden, F., Bosch, J., Kamsties, K., Känsälä, K. & Obbink, H. Software Product Family Evaluation. In: Proceedings of the Third International Conference on Software Product Lines, Springer Verlag: Boston, 2004. Pp. 110–129.

[10] Matinlassi, M. & Niemelä, E. The Impact of Maintainability on Component-based Software Systems. In: 29th Euromicro Conference (EUROMICRO’03), Turkey, 2003. Pp. 25–32.

[11] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P. & Stal, M. Pattern-oriented software architecture – a system of patterns. Chichester, New York: Wiley, 1996. 457 p.

[12] Gamma, E., Helm, R., Johnson, R. & Vlissides, J. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional Computing Series, Addison Wesley, 1994. 416 p.

[13] Matinlassi, M., Niemelä, E. & Dobrica, L. Quality-driven architecture design and analysis method. A revolutionary initiation approach to a product line architecture. Espoo: VTT Technical Research Centre of Finland, VTT Publications 456, 2002. 128 p.

[14] Niemelä, E., Kalaoja, J. & Lago, P. Towards an Architectural Knowledge Base for Wireless Service Engineering. IEEE Transactions on Software Engineering, vol. 31, 2004. 46 p.

[15] Erikson, H., Penker, M., Lyons, B. & Fado, D. UML 2 Toolkit. Wiley Publishing Inc, Indianapolis, Indiana, 2004. 511 p.

[16] Berkenkötter, K. Using UML 2.0 in Real-Time Development: A Critical Review. In: SVERTS: Specification and Validation of UML models for Real Time and Embedded Systems, October 20, 2003. 14 p.

[17] IEEE Recommended Practice for Architectural Descriptions of Software-Intensive Systems in Std-1417-2000. New York: Institute of Electrical and Electronics Engineers Inc, 2000. 23 p.

[18] Object Management Group. UML 2.0 Superstructure Specification, 8.9.2003. 623 p.

[19] Bjökander, M. & Kobryn, C. Architecting Systems with UML 2.0. In: IEEE Computer Society, July/August, 2003. 5 p.

[20] Gardner, T., Griffin, C., Koehler, J. & Hauser, R. A review of OMG MOG 2.0 QVT Submissons and Recommendations towards final standard. In: 1st International Workshop on Metamodeling for MDA, York, 2003. 20 p.

[21] Burt, C., Bryant, B., Raje, R. & Auguston, M. Quality of service issues related to transforming platform independent models to platform specific models. In: Proceedings of the Sixth International Conference on Enterprise Distributed Object Computing, 2002. Pp. 212–223.

[22] Frankel, D. Model-Driven Architecture, Applying MDA to Enterprise Computing. Indianapolis, Indiana: Wiley Publishing Inc., 2003. 328 p.

[23] Christoph, A. Describing Horizontal Model Transformations with Graph Rewriting Rules. In: Proceedings of Model-Driven Architecture Foundations and Applications, 2004. Pp. 76–91.

[24] OMG. MOF 2.0 Query/Views, Transformations RFP, 2002. 32 p.

[25] Shaw, M & Clements, P. Field guide to boxology: Preliminary classification of architectural styles for software systems. In: Proceedings of the 1997 21st Annual International Computer Software & Application Conference, COMPSAC'97, August 13–15 1997, Washington, DC, USA. Los Alamitos, CA, USA: IEEE, 1997. Pp. 6–13.

[26] Jeckle, M. UML Tools, CASE and Drawing.
URL: www.jeckle.de/umltools.html, 6.5.2004.

[27] Microsoft Corporation. COM: Component Object Model Technologies. URL: http://www.microsoft.com/com/default.mspx, 2.11.2004.

[28] SQL.org. URL: http://www.sql.org, 12.11.2004.

Appendix 1: The Q-RDL in Extended Backus-Naur Form

Transformation rules are described by applying Q-RDL rule description language. The syntax of the rules is described below in Extended Backus-Naur Form. The last line (<letter>) presented below is shortened for the sake of clarity.

<newTransformation_stmt> ::= <newTransformation> |
<newTransformation_stmt> <newTransformation>

<newTransformation> ::= <<NEW TRANSFORMATION>>
<sourcePatternName_stmt> <targetPatternName_stmt> <componentInfo_stmt>
<crucialComponents_stmt> <connectionRules_stmt> <<END TRANSFORMATION>>
<sourcePatternName_stmt> ::= <<Source pattern>> <sourcePatternName>
<sourcePatternName> ::= <anyKnownPattern>
<targetPatternName_stmt> ::= <<Target pattern>> <TargetPatternName>
<targetPatternName> ::= <anyKnownPattern>

<componentInfo_stmt> ::= <componentInfo_stmt> <sourceInfo> <targetInfo> |
<sourceInfo> <targetInfo>

<sourceInfo> ::= <<Source information>> <markInfo>
<targetInfo> ::= <<Target information>> <markInfo>
<markInfo> ::= <anyKnownPattern> <anyKnownRole> <anyKnownType>

<crucialComponents_stmt> ::= <crucialComponents_stmt>
<crucialComponents> | <crucialComponents>

<crucialComponents> ::= <<Crucial components>> <crucialElement>
<crucialElement> ::=<crucialElement><<Element>><markInfo> |
<<Element>><markInfo>

<connetionRules_stmt> ::= <connectionRules_stmt> <sourceRule>
<targetRule> | <sourceRule> <targetRule>

```
<sourceRule> ::= <<Source>> <markInfo>
<targetRule> ::= <<Target>> <markInfo>

<anyKnownPatten> ::= <word>
<anyKnownRole> ::= <word>
<anyKnownType> ::= <word>
<word> ::= <word> <letter> | <word>
<letter> ::= a | b | .. | ö
```

Appendix 2: Contents of the Rulebase

Transformation from the Layers architectural pattern to the Blackboard pattern is described below by applying Q-RDL for rule description.

<<NEW TRANSFORMATION>>
<<Source pattern>>
layers
<<Target pattern>>
blackboard
<<Source information>>
layers
component
data
<<Target information>>
blackboard
blackboard
data
<<Source information>>
layers
component
control
<<Target information>>
blackboard
control
control
<<Source information>>
layers
component
computation
<<Target information>>
blackboard
source
computation
<<Source information>>
layers

layer
data
<<Target information>>
blackboard
blackboard
data
<<Source information>>
layers
layer
control
<<Target information>>
blackboard
control
control
<<Source information>>
layers
layer
computation
<<Target information>>
blackboard
source
computation
<<Crucial components>>
<<Element>>
blackboard
blackboard
data
<<Element>>
blackboard
control
control
<<Element>>
blackboard
source
computation
<<Connection rules>>
<<Source>>

blackboard
blackboard
data
<<Target>>
blackboard
source
computation
<<Source>>
blackboard
control
control
<<Target>>
blackboard
source
computation
<<Source>>
blackboard
blackboard
data
<<Target>>
blackboard
control
control
<<END TRANSFORMATION>>

Appendix 3: Contents of the Stylebase

Contents of the stylebase are saved in a text file. Patterns and fields are separated from each other by descriptive tags.

<<--ELEMENT BEGIN-->>
<<Pattern name>>
blackboard
<<Data topology>>
hierarchical
<<Control topology>>
star
<<Diagram>>
composite structure
<<Purpose>>
from mud to structure
<<Abstraction level>>
conceptual
<<Component type>>
computations
control
data
<<Component role>>
blackboard
control
source
<<Connector type>>
messages
<<Attribute>>
modifiability
reusability
extensibility
availability
<<Reference>>
Bushmann et al. 1996
<<Rationale>>
DiSep
<<--ELEMENT END-->>
<<--ELEMENT BEGIN-->>

<<Pattern name>>
layers
<<Data topology>>
hierarchial
<<Control topology>>
hierarchial
<<Diagram>>
composite structure
<<Purpose>>
from mud to structure
<<Abstraction level>>
conceptual
<<Component type>>
varying
<<Component role>>
component
layer
<<Connector type>>
bottom-up notifications
top-down requests
<<Attribute>>
modifiability
portability
reusability
<<Reference>>
Bushmann et al. 1996
<<Rationale>>
DiSep
<<--ELEMENT END-->>

Published by

VTT

Series title, number and report code of publication

VTT Publications 561
VTT–PUBS–561

Author(s)
Merilinna, Janne

Title

A Tool for Quality-Driven Architecture Model Transformation

Abstract

Model-Driven Development (MDD) is about treating models as first class design entities. Model-Driven Architecture (MDA) is an Object Management Group's initiative that proposes to define a set of non-proprietary standards that will specify interoperable technologies with which to realize MDD with automated transformations. The concept of Model-Driven Architecture lies on models at different abstraction levels, where transformations are performed switching between models. Transformations where the abstraction level is changed are called vertical transformations to separate from horizontal transformations where the abstraction level remains unchanged.

Quality-driven model transformation is a horizontal transformation where varying quality attributes of a software product are the driving force for transformation. The quality-driven model transformation relies on the fact that the functionality of the system can be implemented with a wide variety of architectures and therefore with different quality properties. The purpose is to conform to the MDA approach and thus, the goal is to automate the transformation with advanced CASE (Computer Aided Software Engineering Tool) tool.

This thesis focuses on designing and implementing a tool extension that automates the quality-driven model transformation. To accomplish this, a rule description language for defining transformation rules was developed. In addition, a CASE tool evaluation was performed to find the most suitable modelling tool to be extended. Finally, the tool extension was implemented to the Telelogic Tau/Developer.

Keywords
model-driven development, Model-Driven Architecture

Activity unit
VTT Electronics, Kaitoväylä 1, P.O.Box 1100, FI–90571 OULU, Finland

ISBN			Project number
951–38–6439–1 (soft back ed.) 951–38–6440–5 (URL:http://www.vtt.fi/inf/pdf/)			E3SU00217
Date	Language	Pages	Price
March 2005	English, Finnish abstr.	106 p. + app. 7 p.	C
Name of project		Commissioned by	
FAMILIES			
Series title and ISSN		Sold by	
VTT Publications 1235–0621 (soft back ed.) 1455–0849 (URL: http://www.vtt.fi/inf/pdf/)		VTT Information Service P.O.Box 2000, FI–02044 VTT, Finland Phone internat. +358 20 722 4404 Fax +358 20 722 4374	

Julkaisija	Julkaisun sarja, numero ja raporttikoodi
VTT	VTT Publications 561 VTT–PUBS–561

Tekijä(t)

Merilinna, Janne

Nimeke

Työkalu arkkitehtuurimallin laatuohjattuun transformaatioon

Tiivistelmä

Malliohjatun kehittämisen ajatuksena on käyttää malleja ensisijaisina suunnittelukohteina. Model-Driven Architecture (MDA) on Object Management Groupin ehdotus kehittää yleishyödyllisiä standardeja, jotka määrittelisivät keskenään yhteensopivia teknologioita, joita voitaisiin käyttää malliohjatun kehittämisen toteuttamiseen automaattisilla transformaatioilla. MDA:n perusajatus on käyttää eri abstraktiotasoilla olevia malleja, joissa mallista toiseen voidaan liikkua tekemällä transformaatioita. Transformaatioita, joissa abstraktiotasoa vaihdetaan, kutsutaan vertikaalisiksi transformaatioiksi ja transformaatioita, joissa abstraktiotaso ei muutu, kutsutaan horisontaalisiksi transformaatioiksi.

Laatuohjatun mallin transformaatio on horisontaalinen transformaatio, jossa ohjelmistotuotteen muuttuvat laatuvaatimukset ovat transformaation peruste. Laatuohjattu mallin transformaatio perustuu siihen tosiseikkaan, että järjestelmän toiminta voidaan toteuttaa monella eri arkkitehtuurilla ja täten eri laatuvaatimuksilla. Tarkoituksena on pyrkiä noudattamaan MDA-lähestymistapaa, joten päämääränä on automatisoida transformaatio CASE-työkalun avulla.

Tämän lopputyön tavoitteena oli kehittää työkalulaajennus, joka toteuttaa laatuohjatun mallin transformaation. Tavoitteen saavuttamiseksi kehitimme transformaatioiden kuvaamista varten sääntökuvauskielen. Lisäksi teimme mallinnustyökaluvertailun, jonka tavoitteena oli löytää sopiva työkalu laajennusta varten. Lopuksi toteutimme työkalulaajennuksen Telelogic Tau/Developeriin.

Avainsanat

model-driven development, Model-Driven Architecture

Toimintayksikkö

VTT Elektroniikka, Kaitoväylä 1, PL 1100, 90571 OULU

ISBN	Projektinumero
951–38–6439–1 (nid.) 951–38–6440–5 (URL: http://www.vtt.fi/inf/pdf/)	E3SU00217

Julkaisuaika	Kieli	Sivuja	Hinta
Maaliskuu 2005	Englanti, suom. tiiv.	106 s. + liitt. 7 s.	C

Projektin nimi	Toimeksiantaja(t)
FAMILIES	

Avainnimeke ja ISSN	Myynti:
VTT Publications 1235–0621 (nid.) 1455–0849 (URL: http://www.vtt.fi/inf/pdf/)	VTT Tietopalvelu PL 2000, 02044 VTT Puh. 020 722 4404 Faksi 020 722 4374

VTT PUBLICATIONS

543 Holopainen, Timo P. Electromechanical interaction in rotordynamics of cage induction motors. 2004. 64 p. + app. 81 p.

544 Sademies, Anni. Process Approach to Information Security Metrics in Finnish Industry and State Institutions. 2004. 89 p. + app. 2 p.

545 DairyNET - hygiene control in Nordic dairies. Gun Wirtanen & Satu Salo (eds.). 2004. 253 p. + app. 63 p.

546 Norros, Leena. Acting under uncertainty. The core-task analysis in ecological study of work. 2004. 241 p.

547 Hänninen, Saara & Rytkönen, Jorma. Oil transportation and terminal development in the Gulf of Finland. 2004. 141 p. + app. 6 p.

548 Nevanen, Tarja K. Enantioselective antibody fragments. 2004. 92 p. + app 41 p.

549 Koppinen, Tiina & Lahdenperä, Pertti. The current and future performance of road project delivery methods. 2004. 115 p.

550 Miettinen-Oinonen, Arja. *Trichoderma reesei* strains for production of cellulases for the textile industry. 2004. 96 p. + app. 53 p.

551 Hassel, Juha. Josephson junctions in charge and phase picture. Theory and applications. 2004. 38 p. + app. 40 p.

552 Niskanen, Antti O. Control of Quantum Evolution and Josephson Junction Circuits. 2004. 46 p. + app. 61 p.

553 Aalto, Timo. Microphotonic silicon waveguide components. 2004. 78 p. + app. 73 p.

554 Holttinen, Hannele. The impact of large scale wind power production on the Nordic electricity system. 2004. 82 p. + app. 111 p.

555 Rintala, Kai. The economic efficiency of accommodation service PFI projects. 2004. 286 p. + app. 193 p.

556 Kiiskinen, Laura-Leena. Characterization and heterologous production of a novel laccase from *Melanocarpus albomyces*. 2004. 94 p. + app. 42 p.

557 Mäki-Asiala, Pekka. Reuse of TTCN-3 Code. 2005. 112 p.

559 Kiihamäki, Jyrki. Fabrication of SOI micromechanical devices. 2005. 87 p. + app. 28 p.

560 Tuulari, Esa. Methods and technologies for experimenting with ubiquitous computing. 2005. 136 p. + app. 2 p.

561 Janne Merilinna. A Tool for Quality-Driven Architecture Model Transformation. 2005. 106 p. + app. 7 p.

Tätä julkaisua myy	Denna publikation säljs av	This publication is available from
VTT TIETOPALVELU PL 2000 02044 VTT Puh. 020 722 4404 Faksi 020 722 4374	VTT INFORMATIONSTJÄNST PB 2000 02044 VTT Tel. 020 722 4404 Fax 020 722 4374	VTT INFORMATION SERVICE P.O.Box 2000 FI–02044 VTT, Finland Phone internat. +358 20 722 4404 Fax +358 20 722 4374

ISBN 951-38-6439-1 (soft back ed.)
ISSN 1235-0621 (soft back ed.)

ISBN 951-38-6440-5 (URL: http://www.vtt.fi/inf/pdf/)
ISSN 1455-0849 (URL: http://www.vtt.fi/inf/pdf/)

VTT PUBLICATIONS 561 A Tool for Quality-Driven Architecture Model Transformation Janne Merilinna