Inter-level consistency checking between requirements and design artefacts

Final Project
MSc Computer Science

M.J. Looise

March 2008

Inter-level consistency checking between requirements and design artefacts

By
M.J. Looise

Final Project
MSc Computer Science

Graduation committee
Prof. Dr. ir. M. Aksit
Dr. Ir. K.G. van den Berg
A. Gokni, MSc
Dr. I. Kurtev

Chair
Software Engineering

Faculty
Electrical Engineering, Mathematics and Computer Science

University
University of Twente

Abstract

In software development we can identify a number of phases in which a software product is constructed. These phases guide the development from its first conception to realization and maintenance. During these phases, a number of artefacts are created. The artefacts clearly have relations to each other. These relations are bidirectional: a change in, for example, a requirement will impact the architectural design, but also the other way around, changing the architectural design can impact the fulfilment of certain requirements.

It is very important to keep the artefacts consistent. When due to changing circumstances the system needs to be adapted, the requirements of the system will change. Also visa versa, when the architecture is changed, some of the requirements may not be fulfilled any more. In practice, keeping these artefacts in sync with each other is a manual process of reviewing. This is a labour intensive and error prone activity. As a result, the different artefacts can become inconsistent. In order to improve this situation, support for maintaining consistency between requirements and lower level design artefacts in a software development project should be available.

Model Driven Engineering (MDE) techniques can be used to support maintaining this consistency. We propose metamodels for requirements specification and architectural design to explicitly define the structure of these artefacts and to perform consistency checking on them. Metamodels have been derived from a number of research papers, covering requirements and architectural modelling. In composing the requirements metamodel, we only included elements related to the structure of the requirements and its relations to other elements in the requirements engineering process. Elements and relations to artefacts in other design levels are not included. The architectural metamodel is derived from basic components, in an Architecture Description Language (ADL) comparison framework. By performing a case study, we successfully validated these metamodels against concrete instances of requirements specifications and architecture descriptions. This case study checks the conformance of an existing requirement document to the requirements metamodel, and an existing high level design to the architectural metamodel. Using the concepts offered in the metamodels, we can establish trace relations between them.

We propose the use of a separate tracing model to store trace relations between models. A tracing model conforms to the trace metamodel. This metamodel defines typed trace relations with formal semantics. We use interlevel relations, intralevel relations and within-model relations as traces. Currently, two types of interlevel relations are specified in the trace metamodel: the *satisfies* relation and the *refines* relation. We use these relations to check the consistency of both a requirement model with respect to an architectural design, and visa versa. The trace model also offers another advantage: the possibility to model relations in a flexible way. It offers an extension mechanism to include other relations as traces. This becomes possible by defining traces as a role of other relations.

In order to use the proposed techniques and models in practice, we have developed a tool to ensure consistency between requirements and architectural design. In this tool, interlevel relations can be established between requirements and architectural models. Intralevel relations are established between models of the same type. The tool is designed as an Eclipse plug-in. It uses a translation to Prolog, to allow reasoning about the relations. By using Prolog queries to check the formal semantics of the interlevel relations and intralevel relations, the tool enables the user to check the consistency of a requirement document with respect to a certain design.

In this thesis, we have compared our tool to an existing tool offering tracing of requirements to implementations. We can conclude that the tooling in this thesis offers similar functionality, but with a semantic richer way of defining traces. The typed traces, defined in the trace metamodel, offer richer semantics to be used in consistency checking. It also allows for the extension of the consistency checking mechanism. Existing tooling offers lesser possibilities in this respect.

Preface

The work that lies before you is the final piece of my Computer Science study at the University of Twente. I would like to take this opportunity to thank some of the people that have, directly or indirectly made it possible for me to complete this work.

First of all I would like to very much thank my supervisors during my Master thesis, Ivan Kurtev, Arda Goknil and Klaas van den Berg. They have helped me to stay on course, when sometimes I myself was losing directions.
Especially the steady directions and ideas of Ivan have helped me to structure my project and thesis. I will furthermore always have good personal recollections of our many discussions, also those not related to the project. Your door has always been open for me whenever I needed guidance, which helped me very much. Many times I had great doubts about the project when entering your door, but almost always I left with a feeling that we were still on the right track. I hope that you will continue to do the research you like, even when this is not in the beautiful city of Enschede.

I would also like to thank Arda for his comments and vision on the chapters of this thesis. They have certainly helped to present this thesis in the form it is in now. Also the help of Klaas to keep structure and steady directions in the Master project has helped me greatly. The combination of Ivan's steady vision, the structure offered by Arda and Klaas has proven a great combination. I couldn't have asked for a better team of supervisors. Furthermore I would also like to thank the people of Chess, for their time and giving input on my project.

I would of course also like to thank my family and friends to support me during my adventure in the 'tukker-land' Twente. It has been a great experience which would not have been possible without the support of all of you.

I would especially like to thank my mother, who guided me towards the opportunity to continue my study, after having already finished my Bachelor study in the beautiful Zeeland. Even when the situation of leaving home has not been easy. I don't think that I could have finished this study without your support. I would have wished that my father could also have witnessed my graduation on the university. I hope that you will somehow know that I've achieved it. I know it would have made you proud.

I would also like to thank my study friends, which motivated to keep going, also in the periods of difficulty, which all of us faced. I will especially remember the many train and car trips together with my fellow 'Zeeuw', Wouter. Also will I remember the many projects that I carried out, together with the other Wouter. Lastly I would like to thank my room mates for putting up with me, also in the difficult periods of the master project.

Thank you all for making my master study possible.

Atjo!

Martin Looise,

Enschede March, 2008

Table of contents

List of figures

List of tables

List of terms

Architecture	An architecture of software or a computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them [Bass'03]
Requirement	A requirement is a condition or capability needed by a user to solve a problem or achieve an objective [IEEE-610'90]
Requirements phase	A requirements phase is the period of time in a software life cycle during which the requirements for a software product are defined and documented [IEEE-610'90]
Requirement Engineering	Requirements Engineering captures the desired conditions or capabilities needed by a user to solve a problem or achieve a goal he/she has [IEEE-610'90]
Traceability	Traceability is the degree to which a relationship can be established between two or more products of the development process, specially products having a predecessor/successor or master-subordinate relationship to one another; for example, the degree to which the requirements and design of a given software component match. [IEEE-610'90]
Interlevel relationship	An interlevel relation is a relation between model elements and/or models in different development phases within the same iteration of a software development process [this thesis Chapter 4]
Intralevel relationship	An intralevel relation is a relation between model elements and/or models in different iterations within the same development phase of a software development process [this thesis Chapter 4]
Within-model relationship	A within-model relation is a relation between elements in a model in one single iteration and in one single development phase of a software development process [this thesis Chapter 4]

List of abbreviations

ABC	Architecture Business Cycle
AD	Architectural Description
ADL	Architecture Description Language
ATAM	Architectural Trade-off Analysis Method
B2B	Business to Business
CBAM	Cost Benefit Analysis Method
COTS	Common Of The Shelve
CRM	Customer Relation Management
DTD	Document Type Definition
EBNF	Extended Backus Naur Form
EMF	Eclipse Modelling Framework
EMOF	Essential Meta Object Facility
GMF	Graphical Modelling Framework
GORE	Goal Oriented Requirements Engineering
GUI	Graphical User Interface
IDE	Integrated Development Environment
JDT	Java Development Tools
JRE	Java Runtime Environment
KAOS	Knowledge Acquisition in autOmated Specification
MDA	Model Driven Architecture
MDE	Model Driven Engineering
MOF	Meta Object Facility
NWO	Nederlandse Organisatie voor Wetenschappelijk Onderzoek
OCL	Object Constraint Language
OMG	Object Management Group
OO	Object Orientation
PDE	Plug-in Development Environment
QuadREAD	Quality-Driven Requirements Engineering and Architectural Design
RCP	Rich Client Platform
REMM	Requirements Engineering Metamodel
RE	Requirement Engineering
RCP	Rich Client Platform
RS	Requirements Specification
RUP	Rational Unified Process
SRS	Software Requirement Specifications
SuD	System under Design
Synbad	Synthesis based Software Architecture Design Method
TMS	Traffic Management System
UML	Unified Modelling Language
XMI	XML Metadata Interchange
XML	eXtensible Markup Language

1. Introduction

In this chapter, the context of this graduation project will be explained. First the background of the project will be sketched in Section 1.1, to give a better understanding of the importance of this project, and why it is carried out. In Section 1.2, the project description will be given. Here we discuss what topics will be included in this study. After this in Section 1.3, the problem statement will be formulated, which will lead us to the research questions. Section 1.4 will describe the approach that we have taken and the topics that will be treated. The scientific contributions of the thesis will be described in Section 1.5. This chapter concludes with an overview of the thesis.

1.1 Project background

In software development, we can identify a number of phases in which a product is constructed. These phases guide the development from its first conception to realization and maintenance. During these phases, a number of artefacts are created. We can identify the following phases in a software development project: business modelling, requirements formalization, architectural design, detailed design and finally the implementation phase. The artefacts that are created during these phases are clearly related to each other. The business domain for example forms the basis from which the requirements are derived. The architectural design is a solution for the requirements. The architectural design components are created to fulfil certain requirements. Usually, these relations are bidirectional: a change in a requirement will impact the architectural design, and also the other way around, changing the architectural design can impact the fulfilment of certain requirements.

This graduation project is carried out in the context of the QuadREAD project (Quality-Driven Requirements Engineering and Architectural Design). The QuadREAD project is joint research project of the Software Engineering Group (TRESE) and the Information Systems Group at the University of Twente and started in December 2006. The QuadREAD project is supported by NWO (Nederlandse organisatie voor Wetenschappelijk Onderzoek) in the Jacquard Programme. The main goal of the QuadREAD project to achieve a better alignment between analysts who state the requirements, and architects who design the architecture. Alignment between these two fields of expertise is important, because they both concern the same system (only from different point of view). Only when the different artefacts are kept consistency with each other, can these also be used in the development and maintenance phases of the software development.

Model Driven Engineering (MDE) techniques can be used to improve the consistency between artefacts. The way of working of MDE, using metamodels as a core concept for the transformation of models, can be used as a mechanism to check consistency. When stable concepts for both requirements and architectural modelling are known, relations between these two models can be established. These (trace) relations should also be captured in a consistent and precise way.

Next to QuadREAD, the Forest project is carried out by Chess, a software development company and a partner in the QuadREAD project. The Forest project aims at improving the traceability of requirements to lower level design artefacts through a decomposition of a system into subsystems. It offers tool support to assist the automated tracing of requirements to lower level artefacts. In this way the company tries to improve the insight in requirement implementation of a system. The tool can indicate the fulfilment of a certain requirement via a trace link to for example code artefacts. Another goal of the Forest project is the generation of requirement documentation

1.2 Project description

In this graduation project, research will be done at how to store, and retrieve design information into a repository in the form of artefacts. The main research topic in this project is to ensure consistency between these artefacts in case of a change is. Special attention has to be paid to the way that this information is modelled, such that enough information is captured to maintain consistency. For requirements modelling, the abovementioned research at Chess needs to be investigated and possibly be expanded. For capturing architectural design information, the use of Architecture Description Languages (ADL) needs to be considered. An important aspect here is that it should be

possible to model the architecture in a practical way as a part of the state of the practice. Material from current software projects within Chess should provide enough examples from the business domain to validate the models. Special attention need to be paid to the composition of artefacts, and how compositions of artefacts relate to each other.

It should be possible to define relations between artefacts at the requirement level and at the architectural design level. These bidirectional relations should be used to trace changes between the two levels. This can be used to ensure consistency amongst design artefacts. In this project we will focus on maintaining interlevel consistency amongst artefacts. Interlevel consistency checking maintains consistency between elements in models at different development phases. These models are contained within the same iteration of a software development process. The focus will be on consistency between requirements and architectural design.

To show the usability of these concepts in practice, tool support needs to be developed in the form of an Eclipse plug-in. This plug-in should be able to store and retrieve requirements and architectural models and ensure consistency amongst them. To validate the use of these techniques and tools a case study will be carried out with material from Chess.

When we place this research project in the scope of a software development project, it covers both the requirements and the architectural development phase. Both the storing and retrieving of requirement and architectural models will be investigated, in particular maintaining consistency of the interlevel dependencies.

Summarizing, the following topics are studied in this thesis:

- Storing and retrieving early design information, focussed at requirements and architectural design;
  - Modelling of requirements in a semantic rich manner;
  - Modelling of architectural design information;
- Defining consistency between requirements and architectural design;
- Implementing the techniques for the maintenance of consistency;
- Designing and implementing tools to support the maintenance of consistency.

1.3 Problem statement

It is very important to maintain the consistency among the numerous artefacts in a software development process. When due to changing circumstances, the system needs to be adapted; the requirements of the system will change. This will often result in a changed architectural design. Also visa versa, when the architecture is changed, the requirements will also be impacted. In practice, keeping these artefacts in sync with each other is a manual process of reviewing. This is however a labour intensive and error prone activity, and as a result the different artefacts can get out of sync (become inconsistent). The result can be that the artefacts no longer represent the actual system. This is of course an undesirable situation. We aim at a situation in which the relation between requirements and lower level design artefacts can be kept consistent in an easy and tool supported way. Summarizing, the problem we want to address in this thesis is the following: It is difficult to maintain the consistency between requirements and design artefacts in a software development project in case of changes. Accordingly, we can formulate the following research question:

How can we improve the consistency between requirements and lower level design artefacts in a software development project in case of changes?

In this graduation project, we will use techniques from Model Driven Engineering (MDE). This leads us to the following sub-questions:

1. Is it possible to model requirements and architectures using Model Driven Engineering techniques, e.g. by using *metamodels*?
2. Is it possible with these metamodels of requirements and architectural designs to enable *consistency checking*?
3. Is it possible to *maintain consistency* between requirements and lower level design artefacts in case of changes?

4. Is it possible to maintain consistency between requirements and architectural design with *tool support*?

1.4 Approach

To answer the research questions, we first investigate the current work in requirements and architectural models. This allows us to identify the key concepts in both domains. This will be the base for deriving metamodels to support the storing of information in these domains. Once these metamodels are derived, we will validate these models using case studies.

The metamodels will offer a base for deriving trace models that creates relations between requirements and architectures. A trace model will be the key for establishing consistency between requirements and architectural design. By using the relations in the trace metamodel, we can check certain properties of the models connected by a trace model. We will investigate which properties need to be checked before we can conclude that a model is consistent with respect to another model.

After having established the concepts to maintain consistency between levels of development phases, we will demonstrate its usage through the use of the tool support. This tool should support the maintenance of consistency between requirements and architectural models. We will conclude the thesis with a comparison between the tooling based on our work and tooling implemented in the Forest project.

1.5 Contributions

The contributions of this thesis can be summarized as follows:

- An overview of models of requirements based on both state of the practice and state of the art. This means that both theoretical research, but also input from the business side have been investigated. This resulted in a metamodel for requirements.
- An overview of models of architectural design. This again involved the study of current research and practice in this area, especially in the characteristics of ADLs. This resulted in a metamodel for architectural design.
- An investigation of interlevel relations between requirements and lower level design artefacts, in order to maintain consistency between these levels. This resulted in a metamodel for traces.
- A proof of concept of this approach by means of the design and implementation of a tool to support the maintenance of the consistency of relations between requirements and architectural design. The tool has been used in some case studies.

1.6 Thesis Overview

The remaining chapters of this thesis are the following.

In **Chapter 2,** an overview is given on the current state in requirements modelling. The most important components and concepts found in these literature sources will result in a requirement metamodel. Our metamodel is described in this chapter.

Chapter 3 presents an overview of architectural models. It covers the most important concepts in architectural models and how we can arrive at an architecture. Furthermore it discusses a comparison framework for Architectural Description Languages, which will lie at the basis of creating our architectural metamodel. Our metamodel is described in this chapter.

Chapter 4 discusses the concepts of traceability and trace relations. Based on existing literature, we give an overview of the most important elements in a trace. We discuss the characteristics of a trace, and will finally arrive at a definition of how we will treat a trace. We present a metamodel for storing trace information.

Chapter 5 validates the requirements metamodel and architectural metamodel derived in Chapter 2 and 3. For this validation, we used case studies derived from practice.

Chapter 6 explains the tool support for improving the consistency checking between requirements and architectural models. Its design and techniques are described. Also the strong and weak points of the tool are given.

Chapter 7 covers a comparison between our tooling, discussed in Chapter 6, and tool support built by Chess, a software development company participating in the QuadREAD project. This gives insight in the main pros and cons of the technique we used in comparison to an existing technique.

In **Chapter 8,** we conclude this thesis by answering the initial research questions. We also present recommendations and future research.

2. Requirements modelling

This chapter will discuss the modelling of requirements in a software development process. First an introduction to the need of requirement modelling will be given in Section 2.1. After this, the most relevant current work on the area of requirement modelling will be discussed, based on three influential papers in Section 2.2. Section 2.3 will, based on the current research, present a common requirement metamodel to be used as the requirements management metamodel in later chapters.

2.1 The need for requirements modelling

In software engineering, capturing (software) requirements is one of the most important phases in the software development process. Traditionally, Requirement Engineering (RE) is also one of the first phases in a development project. It captures the desired conditions or capabilities needed by a user to solve a problem or achieve a goal he/she has [IEEE-610'90]. It are the requirements that drive a software development project. Without them the project simply would have existed. Loosely translated, requirements of a software project define what the software has to do in order to solve the problems of its users. It is therefore not surprising that requirements are very important (early) design artefacts in a software development project. Only when the requirements correctly resemble the wishes of the end user, can a software implementation really be successful.

Requirement engineering until now has mainly focussed on so called document based requirement specifications [Vicente'07]. This means that the requirements are written down in some kind of textual form. In RE, these documents are called Software Requirement Specifications (SRS). These documents are also officially standardized by the IEEE organisation, like the IEEE-830 standard [IEEE-830'98]. However, next to these more or less formalized models, also other lesser formalized requirements documents are used. This increases the risk that requirement artefacts (requirement lists, use cases etc.) are scattered across a development process without consistency. This use of heterogeneous models can hamper the communication within a project. When for instance separate documents are used to communicate the requirements to both the customer and the software developers, there is great risk of inconsistency between the two documents. Inconsistencies will hamper the communication between the customer and developers, because they are both talking about a different set of requirements. As [Winkler'06] already suggested, requirement artefacts should be kept in one place. To facilitate this, a modelling infrastructure is needed to be able to integrate all the different requirement models in a model centric way and store them centrally, in a consistent way. The aim is to improve requirements management and communication, increasing software quality and lowering costs [Winkler06].

With the increasing tendency to raise the abstraction level in software engineering, and software design in particular, models as an abstraction mechanism are becoming more and more important. As a result, the need to model requirements is also increasing. The main problem with the current state of the practice is however that these models either don't exist or aren't used due to a lack of consensus among them. Only when a proper requirements model can be devised, also a good connection can be made to other design artefacts [Schätz'05].

The need for requirement modelling is even more enhanced with the growing attention on Model Driven Engineering (MDE). In MDE, the main focus lies in performing transformations and maintaining consistency between models of the system. The central entity in this approach is a model of the system. This is back-up by a statement made by Jean Bézivin on the subject of MDE, stating that: "Everything is a model" [Bézivin'05]. This gives models almost the same status as objects have in Object Orientation (OO). Models within MDE are so called first class entities, which are constructed and maintained in similar ways as objects have in OO. In MDE, models are repeatedly transformed to other models to finally achieve a set of models with enough detail to implement in a system. [Kurtev'03].

Lastly, change is another major contributing factor in the need for a proper requirement model. Change is an inevitable fact in software development. One of the most famous statements about change is made by Lehman, in what are commonly known as Lehman's laws of Program Evolution Dynamics. Lehman stated in his first law that "A system that is used, undergoes continuing change

until it is judged more cost effective to freeze and recreate it." [Belady'76] [Lehman'06].The consequence of this inevitable change is that not only the software itself must change, but also the design artefacts (or models) representing the system. Changing the software causes the requirements to change, but changing the requirements will cause other design artefacts (including other requirements) to be effected. A major improvement in propagating these changes on both intralevel and interlevel (discussed later on) would be the use of a single requirement metamodel. This metamodel can provide a common 'language' for the development artefacts to interact with each other.

In the previous section, we mentioned the use of inter and intra level to indicate the way of propagating changes. The distinction between inter and intra levels will be encountered multiple times in this thesis, especially while discussing traceability. For now, we can just state that definitions that use inter level as a prefix corresponds to statements within one level of abstraction. Intra level refers to statements between two different levels of abstraction. The keyword in this discussion is an abstraction. There is no consensus in literature on what an abstraction should be in this setting. In the former section we referred to the level of development (phases) when using the term inter and intra level. We will, for now refer to Intralevel propagation as propagating changes within a development phases. (E.g. propagation of changes from one requirement-document to another). Inter level propagating refers propagating changes between development phases (E.g. propagating changes in a requirement to a design level). We will return to this discussion on the types of relations in Chapter 4.

2.2 Current work on Requirements modelling

This section describes the most relevant work on the area of modelling of requirements. We try to capture as much semantics and structure about a requirement as possible. We will later-on use this information to derive a requirement metamodel, able to represent requirements in multiple ways. This is an important aspect, because industry indicates that a large collection of heterogeneous requirement "models" are used. For instance, a requirements specification can be purely textual or represented as a graphical use case diagram.

2.2.1 REMM Studio

One of the more recent works on requirements modelling is written by Christina Vicente-Chicote et al [Vicente'07]. The main contribution is that the authors try to shift the requirement specification process from a document based to a model based process. To achieve this, a requirements metamodel, called REMM (Requirements Engineering Metamodel), is introduced. This metamodel is used to model the requirements of a system in a uniform way. Based on this metamodel, the authors introduce a suite of tools that enables graphical modelling of requirements, validation of requirements against the metamodel (and OCL), and generation of software requirement specifications (SRS).

Why model requirements

The need for handling requirements in a more formalized way is identified by the authors, and traced back to a number of reasons: First, the growing attention of the software engineering community to Model Driven Engineering is identified as the main driver to requirements modelling. Only when requirements can be modelled, they can really be used in MDE context. This requires the integration of textual requirements documents into models. Next to this, the authors also state that a metamodel could improve requirements reuse and serve as a structured requirements reference model. The metamodel could improve this by explicitly defining the concepts and relationships involved in the requirements engineering process.

Basic requirement concepts.

The authors identify the large number of different requirements models available in literature. They conclude that there is a lack of consensus in the RE community on which concepts and relationships should be identified in a requirement model.

The metamodel in this work is based on an earlier published reuse based RE method, called SIREN. SIREN is a document based and repository based approach. Because the metamodel is based on

SIREN, the metamodel however incorporates some of these concepts. Especially the concept of a catalog as a main entry point for the requirement model is characteristic for this model. The REMM metamodel is however thought to be general enough to use for every RE approach. The model shown in Figure 1 has a reusable repository of requirements, called a catalog. A catalog contains a set of related requirements, called profiles. (e.g. security, integrity etc.) The concept of catalogues is mainly aimed at reuse of requirements (per domain).

Each requirement has a number of basic characteristics, like a textual description of the requirement, a cost and a priority. Next to this, also a type is assigned to the requirement. The types of requirements are based on the well known ISO 9126 quality standard [SSE'08]. The types of a constraint are based on the Volere requirements specification [Volere'07]. The categories as defined by the ISO 9126 standard are however not followed strictly. Also the commonly used taxonomy of Functional and Non-Functional types has been added.

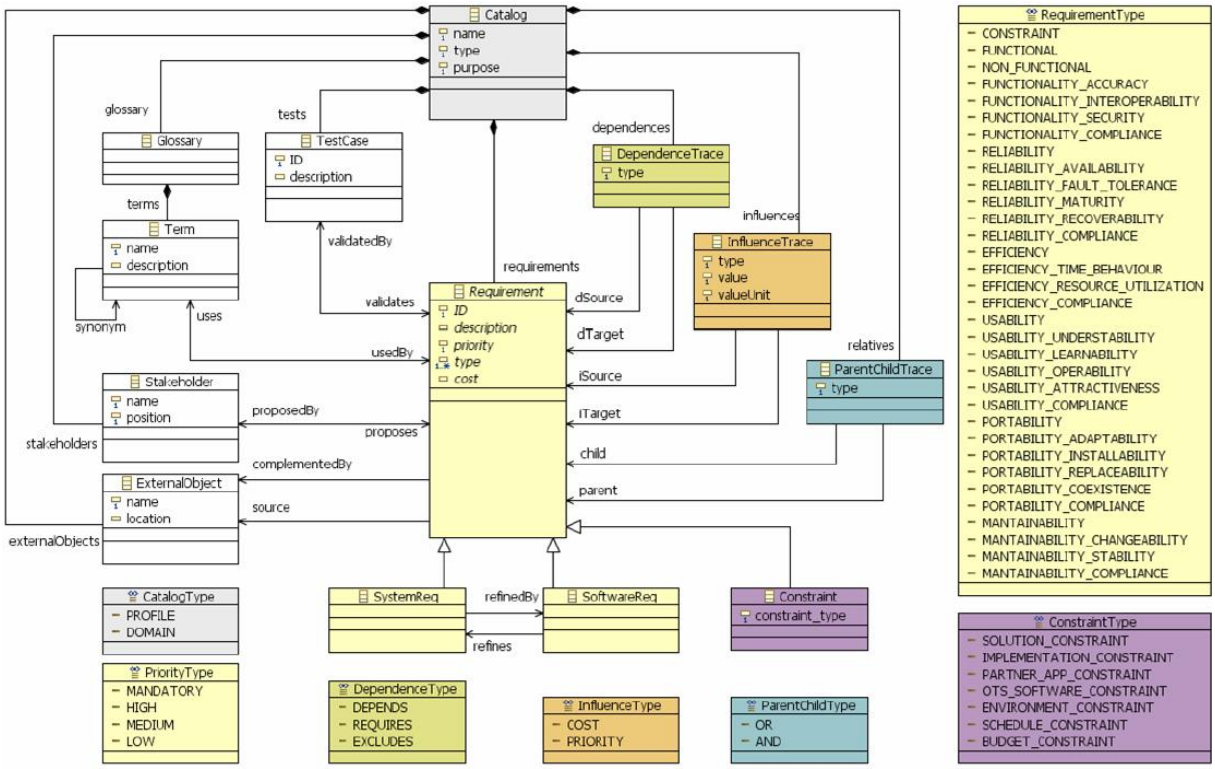


Figure 1: REMM metamodel

Because requirements don't just exist, the model also models the source of a requirement. To do this, both a stakeholder and external object are modelled. Some basic information about the stakeholder is modelled for contact purposes. Next to these stakeholders, requirements can also be extracted from other sources like laws, standards or policies. These are captured in external objects, which can also be used to further explain a requirement or to provide a rationale. These external objects can also be graphical or multimedia files.

The REMM requirements metamodel support the validation phase of the requirement specification process by checking that requirements are consistent and unambiguous. Consistency is maintained by including a consistent glossary of terms. This ensures that requirements related to common terms have consistent definitions. Next to this, ambiguity is checked by defining test cases next to the requirement specification. A requirement is defined as being ambiguous when no test case can be defined for a requirement.

Taxonomies

Each catalog contains a number of requirements which can be categorized with three types of requirements. First, system requirements represent the need of the system, and correspond mostly to business requirements. Apart from these requirements, the model also recognises software

requirements which describe how a system requirement is going to be carried out. Lastly, also constraints are recognised as a special case of a requirement that is mandatory.

Relationships

The REMM metamodel supports a number of relationships to be specified. In selecting the relationships, the authors focused on supporting of the RE process including: elicitation, negotiation, validation and documentation. They split up the relations up into two types; inter and extra requirements traceability relationships. Inter requirements traceability relationships are specified between requirement entities. Extra level requirements traceability relations relate requirements to other non requirements entities (such as stakeholders, files etc.) They call these relations, 'traceability relationships' by following the work of [Pinheiro'03]. Pinheiro specified, based on a definition of [Gotel'94] [Pinheiro'03], a number of taxonomies for traceability. One of these taxonomies differentiates on "Types of involved objects". Hereby, Pinheiro differentiates traces that lead from requirements to other requirements from requirements that lead to other artefacts. Both of these relations are treated as traces. The REMM studio follows [Pinheiro'03] in making this distinction, calling them inter and extra requirements traceability relations respectively.

Inter requirements traceability relationships:

- Between abstraction levels: A *refine* relationship is a relation between two levels of abstraction that refines a system requirement into software requirement. Abstraction levels here refer to the distinction that is made between development phases. System requirements are often specified in an earlier and more abstract development phase than software requirements.
- Between two requirements the following dependency relation are identified:
    - R1 *requires* R2 : R2 is needed to fulfil R1
    - R1 *excludes* R2: R1 and R2 are alternatives to each other, of which only one can be chosen.
    - R1 *influences* R2: The inclusion of R1 will cause a change in the cost or priority of R2
    - R1 *depends* R2: A relation between R1 and R2, not mentioned above.
- Between a requirement and its children (parent/child), in which R1 is the parent of R1.1 and R1.2.
    - R1 *AND* R1.1: To fulfil R1, R1.1 also has to be fulfilled. R1.1 refines the specification of R1.
    - R1 *OR* R1.2: To fulfil R1, R1.2 could be fulfilled, but this is not mandatory. R1.2 is an non exclusive alternative to fulfil R1, apart from R1.1

The parent child relation found in the REMM metamodel resembles the well-known decomposition relation. It is however, slightly different in the fact that they also specify whether the children are of a mandatory nature in order to satisfy the parent.

Extra requirements traceability:

- *ProposedBy*: Each requirement is proposed by a stakeholder
- *ComplementedBy:* A requirement is complemented by an *external object*.
- *ExtracedFrom:* A requirement could be extracted from an *external object*.
- *ValidatedBy:* A system or software requirement must be traced to one or more test cases
- *UsedIn:* A term in the glossary might be used in some system or software requirement.

Tool support

Apart from a metamodel, tool support for requirement modelling is also offered, called REMM studio. REMM studio supports Model Driven Engineering with their REMM metamodel as a basis. Currently, the Model Driven Architecture initiative (MDA) of the Object Management Group (OMG) is the best known approach in the MDE community. One well-known product of the MDA is the top level metamodelling language, called Meta Object Facility (MOF) (a meta-metamodel).

The Eclipse Modelling Framework (EMF) plug-in is one of the most used implementations of the OMG MOF standard. EMF is an Eclipse plug-in which allows designers to create, manipulate and store both models and metamodels. The Eclipse Modelling Framework actually uses a subset of the OMG MOF, called EMOF (essential MOF). The REMM studio is built around the Eclipse framework.

The REMM studio includes two graphical modelling tools (RequirementsTool and CatalogTool), both based on the REMM metamodel. The two tools have been developed using the Eclipse Graphical Modeling Framework (GMF) which is aimed at developing graphical editors using any EMF metamodel, which in this case is the REMM metamodel.

The tools allow users to build textual or graphical models which conform to the metamodel. The *requirementTool* enables depicting requirements and the relationships between them. The *CatalogTool* allows depicting the rest of the elements in the metamodel (stakeholders, test cases, external object etc.). Because the models in the two tools are both based on the same metamodel, the models made in one tool can be used in another. A requirement defined in the RequirementsTool can be imported in the CatalogTool to couple the requirement to a specific stakeholder.

The plug-in allows designers to:
- create a graphical representation for each domain concept in the metamodel;
- define a tool palette for creating and adding these graphical concepts to their models;
- define a mapping between the previous artefacts.

The REMM studio also includes a *validation tool* which checks whether the model is a sensible model. With sensible the authors not only mean conform to the metamodel, but also that the constructions really make sense in practice. The authors state e.g. that, it is possible to create a Influence trace from a software requirement to a constraint, according to the metamodel. This would not make much sense in practice. Checking these extra constrains is achieved using the Eclipse GMF plug-in. With GMF, it is possible to define what a correct model is according the metamodel and according to additional OCL defined constraints.

The REMM studio is aimed at integrating Requirement Engineering (RE) into the MDE approach. In order to satisfy the goal of RE to deliver a textural requirement specification, a document generator is also available. This tool defines a model to text transformation using the Eclipse MOFScript plug-in. To achieve this, a mapping has been made from the REMM model to the IEEE-830 Software requirements specification standard (requirements template). The tool outputs HTML files, but other transformations can also be defined.

2.2.2 Model Based Requirements Engineering for the Web

In 2003, Frank Marschall and Maurice Schoenmakers published a paper giving an overview on the basic concepts of a model based requirement engineering approach [Marshall'03]. It presents a (domain specific) conceptual model aimed on web based business to business applications and its relation to other kind of requirement documents. The model is said to be able to manage requirements of various kind of sources (documents), including B2B standard specifications.

Why model requirements

The authors identify the growing need for requirement modelling in the software engineering community. They identify the reduction of costs and time when using model based development as the main essential factor for its increased need. This reduction of costs and time in development can be achieved by using a common model. Using this common model, properties like completeness or consistency of a model, can be ensured or verified early in development. The authors also identify that models that support the requirement engineering process are currently not widely available. Instead conceptual models and tool support is often restricted to the management of text fragments with their relations. Management or processing of the information contained in these elements is often not carried out. The result of this is a possible increased development time and spending of money. This is mainly caused due to the fact that errors introduced in the first phase of development, are often the most expensive ones. Even when management or processing of requirements information is done, it is often a manual task of verification which is often not practical. A related problem is the diversity of domain and technologic specific terms in the RE process. This is another reason to formalize this information in trying to get a common model that can be used throughout different views on the requirement process.

The solution the authors provide is a common conceptual model supporting multiple views on the same requirements. The proposed model is based on earlier work in the so called UMM-method. They

offer consistency checking through the use of one common conceptual model for all requirement views/documents.

The conceptual model

The conceptual model is tailored for the web enabled B2B applications. As a result, it allows the modelling of business processes and the relations between reused business processes, actors, components and business entities. The following requirements for a conceptual model are identified by the authors:

- The model must be able to model the system under development, but also its domain. This is basically caused by the fact that requirements engineering is started with domain analysis, in order to gather information about the system.
- The model must be able to characterize the system at different levels of abstraction. This refers to a refinement of information which is especially useful in the early development phases (when not all details are yet known).
- For the B2B domain, it should furthermore be possible to refine the system in such detail to incorporate B2B collaborations and communication standards.

The model shown in Figure 2 is the three layered model proposed by the authors. Vertically, we can see a layering based on actors, processes and entities which can be found in every horizontal layer. This makes sense because the horizontal layering is basically a refinement process in which the vertical layers are found on different refinement levels.

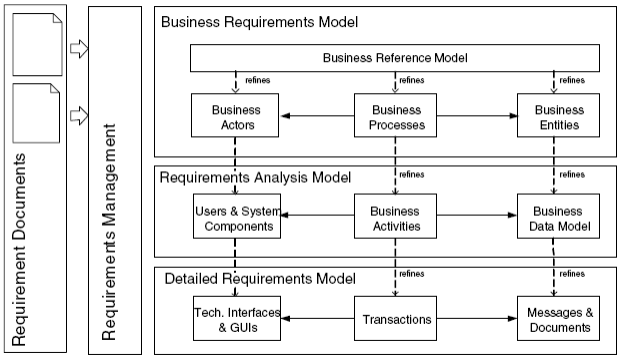


Figure 2: Conceptual model [Marshall'03]

- The business requirement model, as the top layer, models the domain of the system. The top element of this layer is a business reference model which describes a set of common terms and structure for the certain business domain. The remainders of the elements in this layer are used to model the concrete business domain which mainly contains textual descriptions provided by users.
- The requirements analysis model refines the business requirements in such a way that the users, system components, sequences of business activities and a data model are known.
- The lowest level of refinement specifies the system components under development in more detail. The precise level of detail is dependent on the system domain and also on the technologies used. The authors describe a specific B2B domain which is not entirely relevant in this research.

One concept still not mentioned is the requirement management package which plays a central role in this conceptual model. It converts the heterogeneous requirement documents (seen most left in Figure 2) into a common requirements management model (seen as a layered model, most right in Figure 2). In order for this transformation to take place, the authors propose a requirements management metamodel to capture the requirement information of this common requirements management model. The information in this metamodel can is transformed to elements in the conceptual model as described above.

Requirements metamodel

In reaction to the lack of sophistication of current requirement management tools to model requirements, the authors propose a new metamodel. This metamodel is designed to specifically offer

more sophistication than just modelling text fragments with relations between them. This model offers support for the following:

- Tracking involved stakeholders
- Tracking requirement acquisition context
- Management of relationship types between requirements.
- Grouping of requirements by aspects.
- Tracing of the requirement origin and as a result the rationale behind design decisions.
- Tracing of implementation effort (tracing of implementation risks)
- Conflict or dependency tracing in case of change.

In order to relate different requirement document to the proposed requirement management model the authors identify two options:

- Text mining which identifies keywords in requirement documents
- The use of structured forms or diagrams instead of the structured text approach. These forms or diagrams are often very domain specific

The problem is that different concepts extracted from the requirement documents have no relation to each other. Two requirement documents (e.g. a use case diagram and a UMM worksheet) can for example contain two equally named actors. It is however still not sure whether the same actors are meant. In order to make such a relation, a link to the common conceptual model needs to be made.

Integrating requirements into the common conceptual model

The requirements model (see Figure 3) as presented by the authors is roughly subdivided into two parts. One part is used as a pure requirements management metamodel (and offers the capabilities as described above). The other part is used to relate these concepts to the conceptual model (indicated with the grey area).

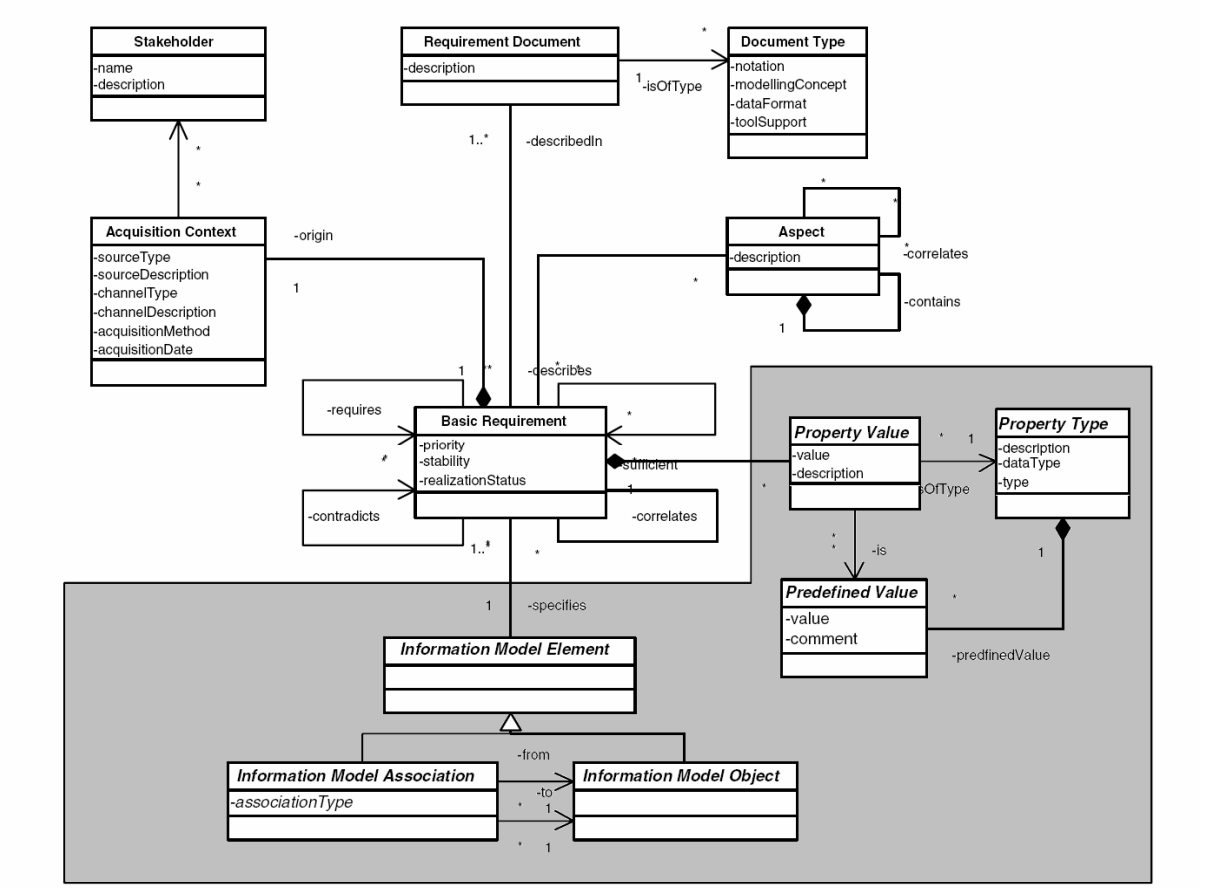


Figure 3: Requirements model [Marshall'03]

The elements of the requirement management metamodel are associated with elements and associations of the conceptual model (grey area). This indicates that a basic requirement justifies the existence of associated elements. Next to the elements of the conceptual model, also property value pairs are used to give the model more expressiveness. Things like quality attributes can be modelled in such a way. The authors have developed a set of predefined quality properties based on the ISO 9126 quality standard. This is meant to serve as a kind of checklist for developers to follow. Using the conceptual model, one can now use the dependencies amongst elements in the conceptual model to discover relationships among basic requirements.

The main problem remains however to extract basic requirements from the different kind of documents. The authors solve this by first defining an abstract syntax for each document type (requirement document). A basic requirement is then defined as a fragment of an instance of that abstract syntax (a piece of a physical document, or a piece of a requirement description). This can be mapped to a set of conceptual model elements together with its properties. The authors use BOTL (Bijective Object Transformation Language) to achieve this mapping. By defining rules that correspond to requirement fragments of the source model, and to elements of the conceptual model one can define mappings for every requirement document to the basic requirements (through the conceptual model).

Once the mapping to the conceptual model is made for every requirement document, tools can process the model and check the requirements documents for consistency. This basically achieves (what they call) inter requirement model checking. If one requirement document is altered, the change can be propagated to other requirement documents, improving the consistency of the models. The authors state that the common conceptual model can be used to improve the communication between engineers by using one common information base. Another advantage is that the common conceptual model can be used as a base for further development steps like architectural modelling and design. Using this model as a base will preserve a relationship from the design models to the requirements from which they are derived.

Tool support

The authors briefly mention that some tool support for this approach is available as a part of the KOGITO project. This tool is aimed at model based requirement engineering that supports development of web based systems. The options of these tools are currently unknown.

2.2.3 SysML

Another well known resource in the field of (software) modelling is the OMG group, and specifically it's modelling language, SysML. SysML provides modelling constructs to represent text based requirements and relate them to other modelling elements [OMG'04]. The SysML group offers a requirement diagram to depict requirements in multiple ways such as graphical, tabular or tree structured. These modelling constructs are said to provide a bridge between requirement management tools and other SysML models.

Basic information

In order to model the requirements, SysML has defined a requirement as a stereotype of UML (Unified Modelling Language) Class subject to a set of constraints. This requirement has a number of basic properties such as identification and a textual requirement. Furthermore, some additional properties can be specified by the user.

Relationships

Next to the basic information of a requirement, the OMG SysML group has defined a number of relationships. These relationships can be seen as both inter and intra level relationships because they not only define relationships between requirements, but also the artefacts in other development phases.

The following relationships are defined:

- Hierarchical requirement relationships: A Hierarchical requirement relationship is a relationship between a requirement and a sub requirement. In this way an entire requirement document can be decomposed into sub requirements forming a tree structured requirement specification.
  Master/Slave or Copy relationship: A Master/Slave relation is a relation between two a master requirements and a slave requirement. A slave requirement is a requirement whose text property is a read-only copy of the master requirement, to be used in a different context [OMG'04]. This is used to achieve requirements reuse in different contexts.
  Derive requirement relationship: A derive requirement relationship is a relation between two requirements, in which one requirement is created by analysing the other. Derivation of a requirement usually involves creating new requirements by analysing a source requirement. E.g. a relation between a requirement on car acceleration, and a derived requirement on the minimal engine power to achieve this acceleration [OMG'04]
- Satisfy relationship: A satisfy relationship is a relation between a requirement and a design or implementation artefact that implements that requirement.
  Verify relationship: A verify relationship is a relation between a requirement and a test case that checks the correct implementation of that requirement.
  Refine requirement relationship: A refine requirement relationship is a relation between a requirement and a model element or set of model elements that elaborate on that requirement. This relation is bidirectional in the fact that it can also support refinement of model elements by requirements.
  Generic trace requirement relationship: A generic trace requirement relation between a requirement and any other model element. The semantics of the relation include no constraints, and is therefore considered weak. [OMG'04] The recommendation of the authors is not to combine this relationship in conjunction with the others.

Next to these relationships, also a rationale construct is offered. This rationale can be attached to both requirements and relationships between requirements to model an explanation of the reason why a requirement or relationship is created.

Taxonomies

The SysML specification also offers a way to introduce a customized taxonomy of requirements through sub typing the requirements. A user of the SysML modelling language can introduce its own categories of requirements as subtypes. By modelling the categories as subtypes, SysML enables the modeller to add constraints that restrict the possible behaviour of the requirement subtype. This makes it possible for a modeller to make its requirement model more domain specific.

Apart form the constructs and relations mentioned above, the SysML furthermore offers multiple ways to represent these elements. A discussion of these graphical and tabular representations is beyond the scope of this work.

2.3. Towards a common requirement metamodel

In this section, we will discuss a common requirement metamodel, created on the basis of the literature discussed in the previous section. This metamodel will purely focus on the structure of requirement and requirement models. It is intentionally kept as simple as possible. We first describe the criteria for selecting the particular metamodel elements. According to these criteria we will discuss the elements that will be included in the metamodel. Next, we will discuss the requirements metamodel itself. After this, we discuss a number of problems we encountered when deriving the metamodel. Because the metamodel will be kept as simple as possible, some possible extensions to this metamodel will be discussed in this section.

2.3.1 Criteria for selecting metamodel elements

When deriving a requirement metamodel for managing requirements, we aim at simplicity as a primary criterion. This means that we try to keep the modelling of the requirements simple and separated from relations to design elements in other phases of the development.

The goal of this thesis is to create relations between requirements and elements in other design phases in order to enable consistency checking between them. These relations will need to be made explicit in a separate model discussed in later chapters. The main reason for keeping these models and its elements separated in this way is the well known concepts of separation of concerns, first introduced by Edsger W, Dijkstra [Dijksta'82]. He stated *separation of concern* as: "This is what I mean by focussing one's attention upon some aspect: it does not mean ignoring the other aspects, it is just doing justice to the fact that from this aspect's point of view, the other is irrelevant". In other words, it makes it possible to focus ones attention to one aspect of the problem. This is also exactly what has been done for ages in computer programming; subdividing a big program is into smaller parts which overlap as little as possible. This makes it possible to solve big problems by first subdividing it into smaller problems (or program parts, like functions, classes, aspect etc) and then solving these smaller problems separately. By later composing these sub solutions we still arrive at the solution to the main problem.

This approach is also taken when trying to answer the main problem in this thesis. This is why we will focus on discovering a model for managing requirements only, and leave any relations to other design level elements. These relations will be treated later-on, as part of a separate trace model.

Concluding we can state the following criteria for selecting model elements of a requirement metamodel:

"Elements of a requirements management metamodel will be related to the structure of requirements and its relations to other elements in the requirement engineering process, not to other elements included in other design levels"

2.3.2 Basic requirement concepts

Before we can discuss what information should be captured in a requirement metamodel, we should first form consensus on the term requirement itself. We will use the definition of the IEEE, provided in the standard glossary of Software Engineering terms [IEEE-610'90].

When we talk about a requirement, we mean: "(1) A condition or capability needed by a user to solve a problem or achieve an objective. (2) A condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed documents. (3) A documented representation of a condition or capability as in (1) or (2)". It can be stated that instead of document we can also read model in this context.

When analyzing the definition as stated above, we can see that a requirement involves a description of a *condition* or *capability* which needs to solve a problem. It also includes a *user* which wants this problem to be solved. A problem is the difference between the current and desired situation.

Requirement element:

From the abovementioned definitions we can deduct that a requirement model should contain at least a requirement element with *a textual description* of the (future) capability of the system. This description will describe a gap between the current and desired situation (problem). The existence of a textual description is backed up by all the models discussed in the previous section. Besides capabilities, the definition also mentions conditions. These are often considered as *constraints* that must hold in order for a system to function. This is not necessarily information that is contained in the requirement element itself and can be modelled in separate entities.

Each requirement must also be uniquely *identified* and have a *name*. The identifier can be used by systems to uniquely identify requirements; the name is more aimed at the human reader/user to recognise the requirement.

Another common attribute of a requirement among the discussed literature is a *priority* of a requirement. Both the REMM and the B2B based models (we will use this term to refer to the work of [Marshall'03]) use this concept, but have assigned different values to it. In order to be independent of both models, we use the MoSCoW method to indicate the importance of a requirement by the customer. (Must have this, Should have this if at all possible, Could have this if it does not affect

anything else, Won't have this time but Would like in the future). This classification is taken from the software development methodology DSDM, and has been proven in the past [Ash'07].

The REMM metamodel also mentions a *cost* attribute assigned to a requirement. This field is not intended to carry exact information about the cost of implementing a requirement, but to capture the designer's intuition. It is important to capture this information, in order to support the decision making process of which requirements need to be implemented (first). We will further discuss the influence of the cost element when we discuss the relationships in the metamodel.

Inspired by the work of Marshall [Marshall'03], we will also offer the possibility to add (user defined), tangible quality properties to a requirement. This enables us to specify (none standardized) quality properties. Specific quality attributes like costs, speed or stability (Some also found in the REMM and B2B models) will have to be modelled. In order for these quality attributes to be tangible (and be able to measure them later on), we need a way to give values to them. We can do this by using a similar construction as seen in Figure 3 (Using a property/value construction). Because these quality properties are usually imposed by so called Non-Functional Requirements (discussed later on), these properties should only be used to make Non-Functional Requirements tangible.

Stakeholder:

Another common element in the requirement models discussed earlier is that a requirement always has a user involved. In RE, this user is known under a more common name; a *stakeholder*. A stakeholder is more than just a user of a system; it also includes other people who are in another way involved in the development process. The problem with the concept of stakeholder is that there is no consensus on the meaning of the concept [Sharp'99]. People from a business perspective often mean different things with a stakeholder than people from information systems, or software engineering. Another contribution to this confusion is that the term stakeholder also differs depending on which requirement is modelled. A business requirement involves more stakeholders, including those affected by non software systems, then a software requirement. A software requirement often includes people only affected by the software system. We will take a definition from the software engineering field that doesn't exclude either of these stakeholders. To do this we will take the definition from [Kontonya'98] in the work of [Sharp'99], defining a stakeholder as: "S*ystem stakeholders are people or organisations who will be affected by the system and who have a direct or indirect influence on the system requirements".* Note that a system here doesn't necessarily need to be a software system.

In the work by [Marshall'03] a stakeholder is modelled as a part of an acquisition context in which the requirement is introduced. This specifically binds a stakeholder to an acquisition context. For reasons of simplicity we follow the way of working of the REMM studio metamodel, and model the stakeholder as a separate entity related to a requirement.

One last reason to model a stakeholder in the requirement model is to be able to know who is affected by the requirements. When conflicts or uncertainties about a requirement occur, the original stakeholder(s) can be tracked down for consultation. This is why contact information of the stakeholder is important to capture.

Rationale:

In order to capture why a requirement is introduced, a rationale for a requirement is needed. The need for such an element is identified in all the discussed literature. Next to a simple explanation of, we will also use this element to model other external elements in the domain, such as sources and laws (identified in the work by [Vicente'07]). A rationale furthermore always has to be related to a stakeholder who introduced the rationale.

External entities:

A highly related concept to the above-mentioned design rationale is what can be called an external entity. These entities can be (legal) entities which relate to requirements. As indicated by [Vicente'07], an external entity can represent a law or guideline from which a requirement is derived. Because they do not introduce requirements themselves but they are related to a requirement, they are often modelled separately. A rationale as described above will link to these external entities as sources or

reference. [Marshall'03] also recognises the need for such entities, based on the need to maintain information about requirement acquisition.

2.3.3 Requirement taxonomy

The previous discussed literate all use a different taxonomy to classify requirements. The REMM metamodel, uses a taxonomy, created by making a distinction in three kinds of requirements: system requirement, software requirements and constraints. These concepts also reoccur in other literate covering requirement engineering. These first two kinds of requirements are also often called business and functional requirements respectively. Another often made distinction in the REMM metamodel is between Constraints, Functional and Non-Functional Requirements. Lastly we also see that quality properties often have a role in the requirement taxonomies (both in the REMM metamodel and the B2B based model). In this section we will shortly discuss these taxonomies, in order to come up with our own taxonomy to use in the requirements metamodel.

Business and Software Requirements

When looking at the goal of this thesis; consistency checking between requirements and design artefacts, we usually refer to consistency checking between functional requirements and architecture, not between business requirements and architecture. This does however not mean that business requirements are not important. In the REMM metamodel, System or Business requirements are described on a different abstraction levels than software requirements. They are related by a refinement relation. This is a good reason to also make a distinction between these kinds of requirements in a requirement metamodel. We will therefore model the Business requirement as a special case of a requirement element. A business requirement will have a refinement relation with a software requirement.

Using a refine relation between business and software requirements is a support the natural way of problem solving, often found in software development. As mentioned in [Wieringa'03], a design activity is often partitioned into layers. Wieringa, in his work suggest three aggregation levels; the composite system, consisting not only out of software, the System under Design (SuD), and the decomposition of the SuD [Figure 4].

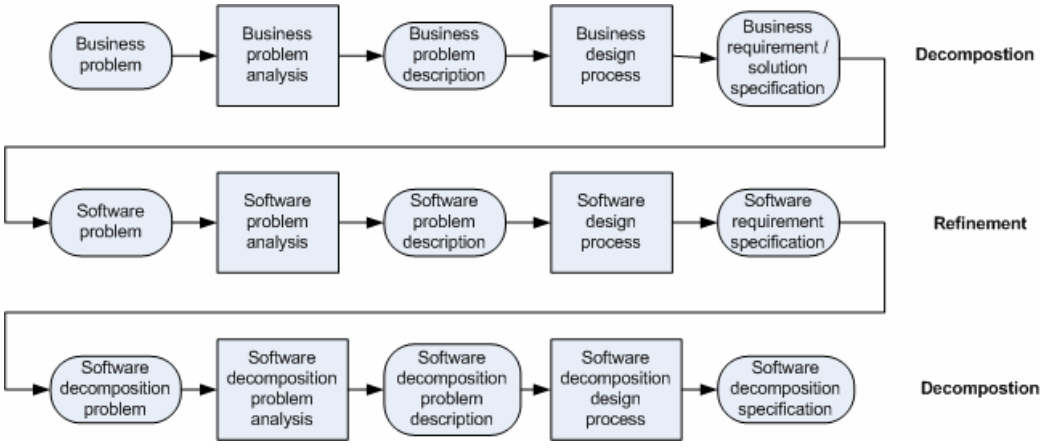


Figure 4: Three aggregation levels [Wieringa'03]

The Composite system can be considered the business in which a software system operates. Its requirement should also be modelled on the level of the composite system; basically modelling requirements on the business level. An implementation of such a business requirement need not be a software system. This can also be a solution of organisational or other physical nature. It is possible that a business problem, stated in a business requirement, is solved by software. This software will have its requirements stated as software requirements. Here the business requirements will be connected to the software requirement. We can best illustrate this with an example:

Consider a company with a manual registration system for managing its customer details in the Customer Relation Management department (CRM). The company's management has identified that

the CRM business unit is costing the company too much money, and states the business requirement that the spending level of the CRM department needs to be decreased. This can however be achieved using different solutions. One solution might be to limit the maximum amount of time spent per customer on product advertisement. Another solution could be to downsize the number of employees working in the department. As IT professionals, we however know that these kinds of departments are also very suitable for making use of an automated CRM management system. One solution could as a result be to start using an automated searchable customer registration system, and thus limiting the time (and money) searching for the right customers. This would be the top level software requirement born out of the business requirement to decrease the spending level of the CRM department.

What we see from the abovementioned example is that we can discover a general problem solving process in which a problem is subdivided into different solutions [see also Figure 5]. These solutions can also be seen as problems on a different level. The CRM management system of the abovementioned example is for example a solution to the business domain, but a problem for the software architects to build. Solutions of this architect will in turn also be problems (for decomposition in smaller parts). What makes these solutions turn into problems on a different abstraction level is the domain in which they reside. Solutions in the business domain are problems in the software domain. Solutions in the (high level) software domain are problems for the software decomposition domain. This is also clearly seen in Figure 5.

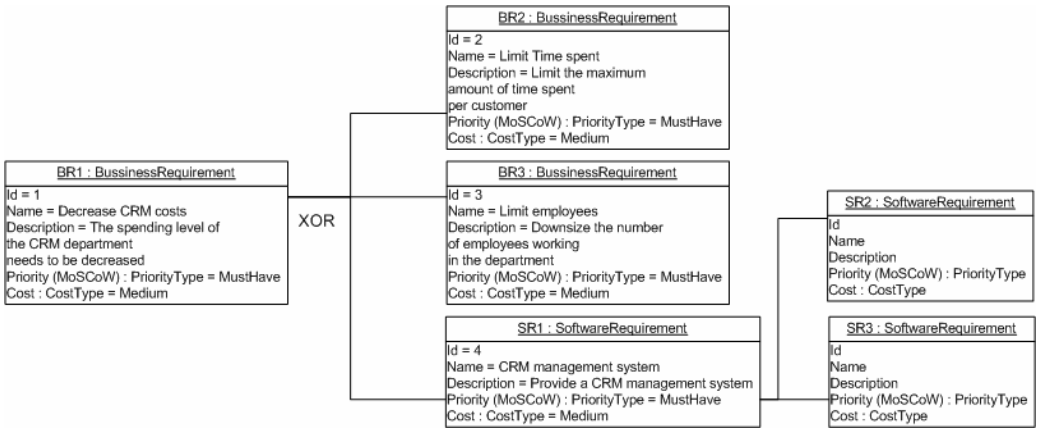


Figure 5: Refinement of Business and Software requirements

To be able to support this refinement of software requirements out of business requirements, we will make the distinction between business and software requirements in the requirement metamodel. Hereby, we can basically define a natural problem solving process in which a business problem is refined into solutions which can possible be solved by software. The software requirements will usually be decomposed into smaller sub requirements until they are small enough to be solved by simple software units. This last decomposition is also supported by the aforementioned well known concept of separation of concern. Decomposition of requirements into smaller (sub) requirements is discussed later on when we discuss requirements relationships.

Function & Non-Functional Requirements.

Another common taxonomy makes the difference between Functional and Non-Functional Requirements. A problem with these concepts is that they often hint at a certain priority, leaving the Non-Functional Requirements with lesser priority then Functional requirements. This is of course not true, and not the intention of the distinction. The difference between the two is that, Functional requirements describe behaviour of the system to support user goals, tasks or activities (similar to the definition stated by the IEEE). Non-Functional Requirements however include constraints and qualities [Malan'01]. Qualities are also stated as properties or characteristics of the system that the user cares about [Malan'01]. Constraints are seen as non negotiable requirements.

We will also make the distinction between Functional and Non-functional requirement in our metamodel. One reason for making this distinction is because the information we want to capture in the two kinds of requirements is different. Non-functional will often be made as tangible as possible to enable measurement and automated checking of these requirements. This, requires more than a

textual description of a requirement, and would benefit from a separate construction to capture this information. We will include a similar constriction, like done in the B2B paper, to capture this information in a property/value pair. On order to only relate this property/value pair to requirements capturing quality attributes, we will make the distinction between Functional and Non-Functional requirements.

Requirements / Constraints

We can deduct from our definition of a requirement, that constraints are treated different from requirements. Because constraints and qualities can both be considered Non-Functional Requirements, modelling properties of the system on respectively a mandatory and non-mandatory level, these entities should be modelled separately. Constraints are closely related to Non-Functional Requirements, as described by [Malan'01]. We will introduce a separate model element for constraints to contain a mandatory qualities attribute. Because constraints and Non-Functional requirements both describe quality attributes, they are highly related. Constraints are therefore seen as a special case of Non-Functional Requirements. We will define a constraint as a mandatory Non-Functional Requirement. In this way it is also possible to use the property/value pairs to specify a constraint.

Quality taxonomy

In the REMM metamodel, we see that a type attribute is used to indicate which quality attribute of the ISO9126 standard the requirement describes. This classification is furthermore amended with a functional, non functional and constraint type. Because we have already suggested having separate model elements for these last requirements types, there is no reason to make this distinction in a type attribute.

When the need arises to model the quality attributes that a requirement addresses specifically, this is also be possible using the property/value pair. We therefore suggest not to make a separate taxonomy like done in the REMM metamodel using a RequirementType. This leaves freedom for the modeller to choose its own quality taxonomy for Non-Functional Requirements when needed, (through the use of the property/value construction). Moreover it doesn't overcomplicate the model with unnecessary details.

Other taxonomies:

A last taxonomy, seen in the work by [Marshall'03], is grouping by aspects. The work of [Marshall'03] does not offer a very detailed explanation of its function, except that it offers a grouping mechanism for requirements with similar goals. Because we will primarily focus on the structure and relations among requirements, and not on support of aspect classification we will not include this classification in our metamodel.

2.3.4 Relationships

Another important aspect of a requirement metamodel are the relations that can be established between the requirements. As discussed earlier we will purely focus on relations within the same development phase, limiting our relations to relations between requirement and relations that clarify or extend requirements. In this section we will discuss which relations will be included into the requirement metamodel. We will, for now, maintain the same distinction as the work by [Vicente'07], using the terms inter and extra requirements relations.

Inter requirements relationships:

When we filter the relations found in the abovementioned literature, purely between requirements, we come up with the following relationships between requirements.

Decomposition

To be able to form a hierarchy of requirements, seen in requirement specification documents, a decomposition relation is needed. This relation makes it possible to model a part/whole construction

between requirements. It should also be possible to build more complex decomposition relations, as seen in the work describing the REMM metamodel, in order to indicate fulfilment criteria.

An interesting question when defining a decomposition relation among requirements is; when is a requirement composition as a whole is satisfied? (When is the requirement composition met?) Traditionally a whole is only satisfied when all its parts are also satisfied. This kind of decomposition is not only found in the aforementioned REMM metamodel, but also in work by others. In the work by [Antón '96], describing goal based requirement analysis, goal hierarchies are built using a decomposition of goals into sub goals. Here a goal is only achieved when all its sub goals are achieved. The authors call this kind of decomposition relationship the 'AND' decomposition relationship, and it is assumed to be the default decomposition relation. As an addition, they also mention the use of an 'OR' variant on this relation. Here the goal is achieved by either achieving sub goal A or sub goal B.

The REMM metamodel also uses these keywords to distinguish between two kinds of decomposition relations, in which the AND variant is similar to the one used in [Antón '96]. The AND variant in the REMM metamodel is also called refinement, because the parts will add more detail which need to be fulfilled, to fulfil the parent (or whole). Usually such decomposition will be used to make requirements more detailed. It can be used to support the aforementioned principle of separation of concern. The smaller sub requirements are easier to map to implementation elements than big requirements often lacking detail.

The OR variant is slightly different in both papers describing this relation. The REMM paper describes this as optional for fulfilment of the parent. It is mentioned as an alternative way to fulfil the parent. The work by [Antón 96] however mentions the OR variant as being a relation in which either of the children can be sufficient for the fulfilment of the parent, but it is not optional.

When we look closely at the decomposition relations mentioned in the work of [Antón 96] and the REMM metamodel, we see that the main difference lies in the mandatory nature of its fulfilment.

To make matters even more complicated another kind of decomposition relation, used in requirement modelling, is the XOR relation. In the work of [Bruin '01], relation between requirements and (possible) solution fragments are established. In his work, de Bruin describes the usual AND/OR decomposition, similar to the ones already discussed. What is different however is the use of an XOR variant, to select exactly one constituent (or child) [Bruin '01]. This makes it possible to model requirements of which exactly one of the possible options should be chosen. This is especially useful when modelling alternative requirements.

Having discussed existing literature surrounding requirements modelling we can see the need for the decomposition relation together with its AND and OR variant. Because also the XOR variant has its purpose of existence when modelling (exclusive) alternatives, we will also include this relation when composing a requirement model. Therefore we include the following types of decomposition:

- AND; used to indicate that all the parts need to be fulfilled to fulfil the whole. Here the parts are actually refinements (e.g. an addition) of the whole, which needs to be fulfilled.
- OR; used to indicate a possible refinement. Now the parts need not be fulfilled in order to fulfil the whole. (can also be seen as suggestions)
- XOR; used to indicate that one of the possible parts needs to be chosen for the whole to be fulfilled. Fulfilling multiple parts is however not possible.

These decomposition types enable the requirement modeller to build more complex requirement document. Figure 6 gives an example of such a requirement model:

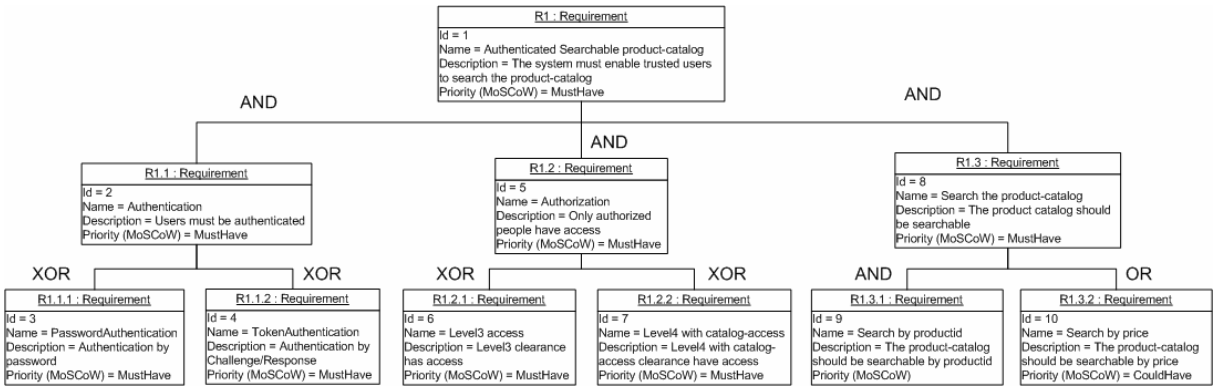


Figure 6: Decomposition example

This requirement documents described an administrative system which enables users to search through a product catalogue. What we see here is that authentication, authorization and a search option are mandatory. In both authentication and authorization we see that alternatives have been modelled. We also see that searching must be able to be performed based on product information. An optional search method is also modelled based on price information.

Influence:

An influence relation is established when one requirement changes the attributes or characteristics of another requirement. In the work by [Vicente'07] we see two of these influence relations:

- Cost: Implementing one requirement changes the costs of another requirement
- Priority: Implementing one requirement changes the priority of another requirement.

A big problem when making statement about the influence of costs and priorities between requirements is that they are often supported only by the intuition or experience of the person designing the requirements. One can state that a requirement, 'people should be able to access the system remotely', has both a cost and priority relation with requirements making statements about security. Most software engineers will confirm that this relation exists. The question is however, why do software engineers believe that such a relation exists? Most probably this is caused by past experience or the knowledge when building software systems.

In reality, software engineers are making (unconscious) assumptions about the implementation that will satisfy this requirement. The fact is that only after an implementation of a certain requirement has been chosen, statements about cost or priority relations can be made. It might turn out that even when a cost influence relation seems to exist between certain requirements, the chosen software solution will not imply this relation. Only when decisions on the chosen solutions are taken, we can with certainty create influence relations in the area of cost or priority (Also see Figure 7).

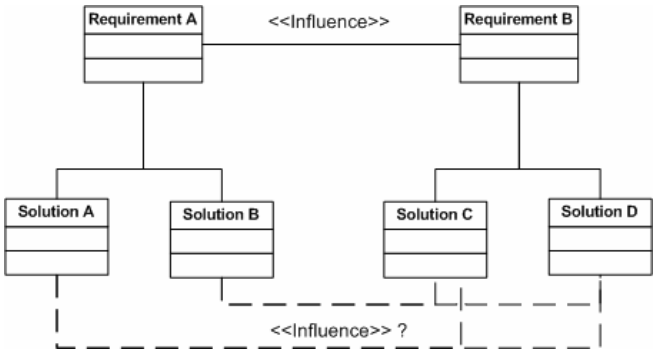


Figure 7: Requirement / Solution dependency

We have to therefore make the remark that when using the influence relation among requirements, this it is merely used to capture the intuition or experience of the designer. Ignoring this information would be unwise, because it can help a software designer make decisions about the possible implementations. We will therefore include this relation in our metamodel.

Dependency

Following the REMM metamodel, we see that a number of dependency relationships can be found. This relationship element does not need any additional attributes to give meaning to it, and can therefore be used for a number of dependency relations between requirements. We will use this relation for a collection of relations found in the aforementioned literature.

- Excludes: If one requirement is chosen, the other in the relation can't be chosen. This relation is closely related to the XOR relation found in the decomposition relation. The XOR decomposition relation is however meant to model exclusive alternatives amongst a decomposition of a certain requirements. The Exclude relation is different in the fact that it need not be bound by a decomposition structure. This relation has close resemblance to the 'contradicts' relation found in the work by [Marshall'03]. Because a description of this relation is not included, we assume it to serve similar goals, and use the term exclude instead.
- Requires: One requirement is needed to fulfil the other. This again closely resembles the decomposition relation, because of its fulfilment relation to other requirements. In the AND variant of the decomposition relation, we also see that one requirement (the parent) is only fulfilled when others are too (the children). This relation is not bound to the decomposition context, and can relate requirements otherwise not related by decomposition.
- Depends: Like in the REMM and SysML works, a general relationship is created to capture any relationships between requirements that can not be captured by the above.

Refine

This relationship is found in both the REMM metamodel and the SysML specification, but has different meanings in both. In the REMM metamodel, the refine relationship is used to define a refinement of a business requirement into a system requirement. In the SysML specification it is used to improve the detail of a requirement through the use of models of that requirement. What we can see by looking at these two approaches is that there's a need for both of them; refinement of a requirement with a new abstraction level, and refinement through the addition of details.

One problem when trying to capture the refine relation in one concept is that people often have their own vision of the concept in mind. We need to develop a clear definition which people will recognise as being natural to the already known term. The dictionary uses the word 'refinement' to indicate a more detailed version or development of an element. Both the uses by the REMM metamodel and SysML fulfil this concept, so both can really be seen as refinement.

We have however already seen the term refinement before, when discussing the decomposition relation, specifically the AND variant of the decomposition. This relation basically covers the need of adding more textual details as described by the SysML specification. We will therefore use this concept of decomposition to make a more detailed specification of a requirement into smaller sub requirements. When other (external) models are used to describe a requirement, the external entity concept could also be used to capture the information. The SysML interpretation of the refinement relation as a result need not introduce a new concept in the requirement model.

The use of refinement to model a relation between a business requirement and a software requirement can also be seen as a kind of decomposition. A software requirement doesn't just exist, instead it usually has a business driver behind it, which in turn has a problem as a background (described in a business requirement). Often, business requirements will as a result be decomposed into a number of software requirements, which all need to be fulfilled in order to fulfil the business requirement. One can however also imagine a situation in which (exclusive) alternatives software requirements are presented to satisfy a business requirement. Because the decomposition relation is perfectly suited for these situations, we therefore choose to use the decomposition relation for this kind of refinement.

The B2B based model uses yet another form of refinement; refinement of requirements into other design level artefacts. It is not the goal of this requirement model to refine requirements into other design artefacts (in other development phases). This will be investigated further in the trace model and will therefore not be treated here.

One final aspect which is often related to refinement is the change of a requirement through time. This can for example be caused by new information on a requirement. In this case, a decomposition relation is not a very well suited solution to model this, because we do not want to introduce multiple versions of a requirement. We basically want to create a new version of a requirement which replaces the old requirement. Every relation and attribute of the old requirement will in the ideal situation be transferred to the new requirement and possibly adjusted. How to implement the maintenance of consistency when introducing a new version of a requirement is outside the context of this chapter, and needs further research. It should however be possible to achieve this. Two possible options are demonstrated in Figure 8.

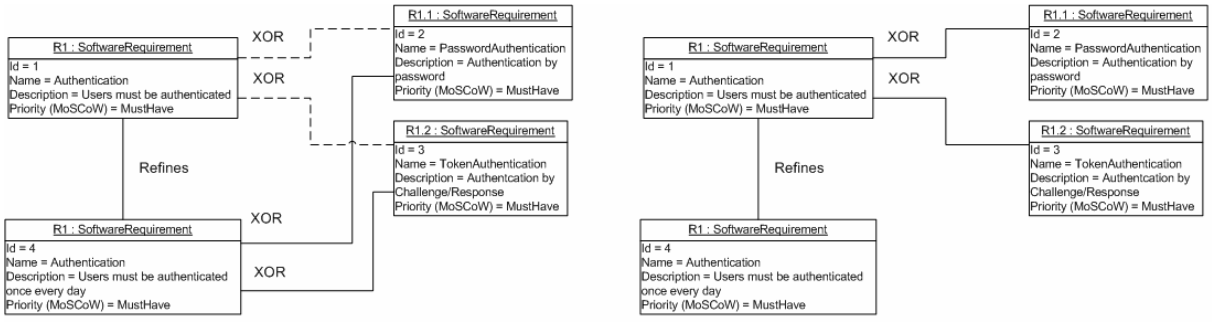


Figure 8: Refinement changing and maintaining the original relations

In Figure 8, we see that the relations of the original requirements are derived by the new version (left). We also see another option, in which the new version of the requirement has not derived the relations (right). In both versions, it's possible to detect that the XOR relations existed and still exist in the new situation.

What is clear is that we do not want to throw away the old requirement when a new version is introduced. It can for instance be the case that this old requirement is still needed to improve the understanding of the change, or to be able to roll back a changed requirement later on. To enable the modelling of this situation we define the refine relation as a relation between two requirements in which no two requirements can have the same source requirement. A requirement can only have one refinement relation. A refine relationship can furthermore only be established between two requirements of the same type.

Affects

To also enable the modelling of quality or constraint of a requirement, we also model a new kind relationship which we call the affects relationship. This relationship has a requirement as a source, but always has a non function requirement (quality property or constraint) as a target. In this way one can model the qualities of a system and/or constraint a system.

Extra requirement relations between related elements on same development phase:

Next to relationships between requirements themselves, we can also identify some relationships between requirements and elements containing information about the requirement (still on the same development level):

ProposedBy

As discussed earlier, every requirement has stakeholders. This relationship relates to one stakeholder in specific, the stakeholder that introduced the requirement into the requirement specification process.

ComplementedBy

When a requirement is complemented by an external object to further explain or describe the requirement, this is captured in an external object. This relation makes it possible to relate the requirement to this concept.

ExtractedFrom

A requirement can also be related to an external object in a different way. In this case it is related because the requirement is extracted from the external object. In this way it can be used in order to clarify a requirement in case of uncertainty or discussion.

2.3.5 A Converging common requirement metamodel

After having discussed the elements that will be included into a common requirement metamodel, we are ready to introduce a common model based on these elements. The common conceptual metamodel based on the abovementioned literature and model elements is depicted in Figure 9. This model will be used as a metamodel to manage the requirements.

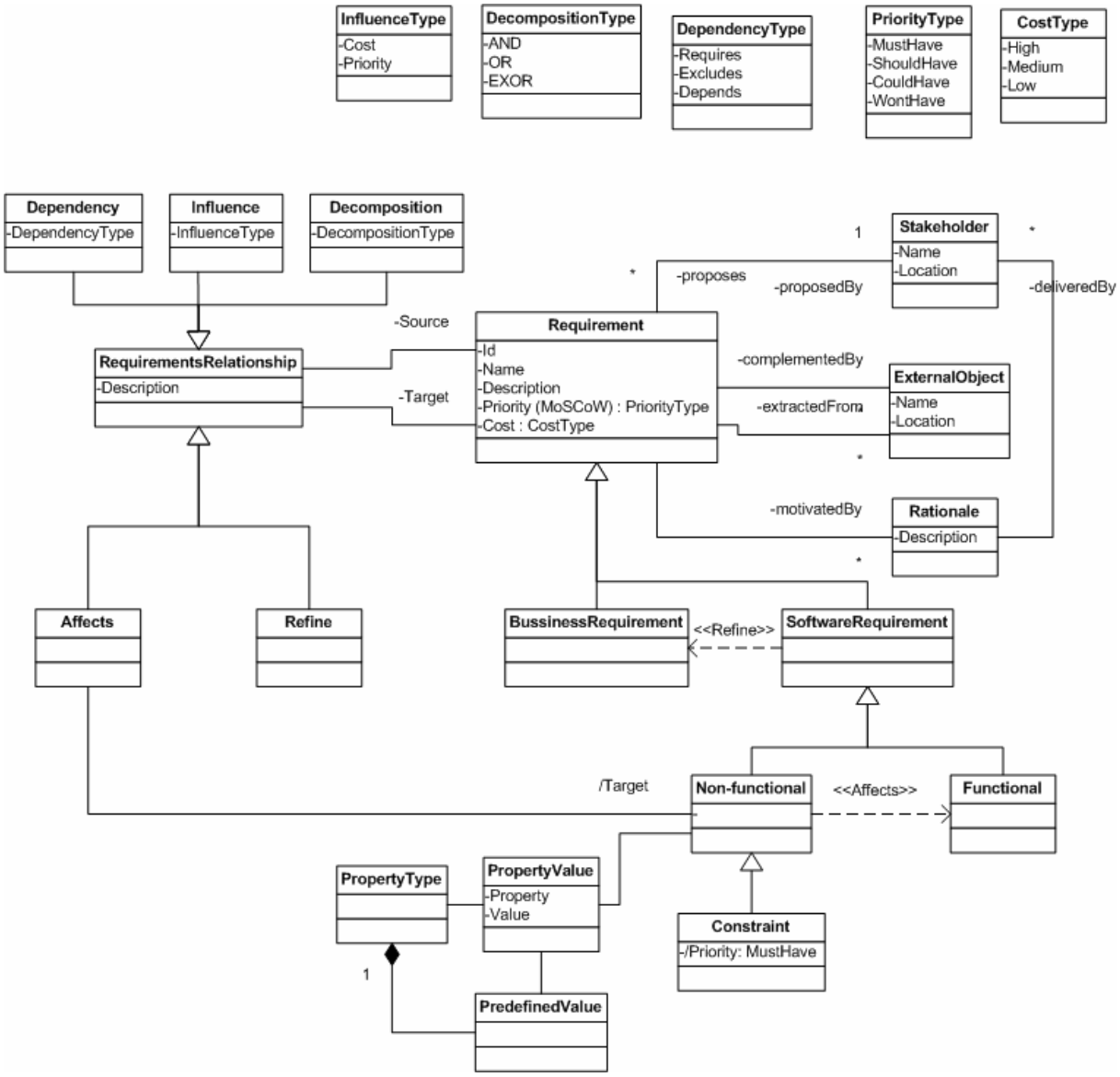


Figure 9: Common requirement metamodel

The model basically has only a few basic concepts which are requirement, relationship, stakeholders, external object and a rationale. As a result, the model is kept simple enough to be used in a requirement management setting, and also has enough power to capture the structure and relationships among requirements. In this section we will define each element of the model in a point by point manner.

Basic requirement concepts

- Requirement: A requirement is a condition or capability needed by a user to solve a problem or achieve an objective [IEEE-610'90]. A requirement has an identifier, a name, a descriptive text in which the requirement is explained, a priority according to the MoSCoW classification and a cost capturing the designer's intuition about the cost.
- Stakeholder: Stakeholders are people or organisations who will be affected by the system and who have a direct or indirect influence on the system [Sharp'99]. The stakeholder element describes the stakeholder who proposed the requirement. It contains contact information of the stakeholder to enable communication about the requirement.
- ExternalObject: An external object is an entity related to a requirement capturing details of the requirement. This element contains extra information to explain a requirement through the use of external objects/files. This element can be used when a requirement originated from an external object like a law, policy etc. This element can be used to save this information.
- Rationale: A rationale is the reason why a requirement is introduced. The rationale is linked to the stakeholder who delivered the rationale.

Taxonomy

To resemble the separation of requirement as either a business or a software requirement, two specializations have been made to differentiate between the two. This is used to couple the business domain to the software domain. Next to this, we separate a software requirement into either a Functional or Non-Functional Requirements. Constraints also have a different meaning in the fact that they have a mandatory nature. Both Non-Functional Requirements and Constraints can be clarified through the use of quality attributes, maintained using the property/value elements.

- Business Requirement: A business requirement is a requirements originating from the business domain. These requirements describe the requirement of the business, not necessarily satisfied by software. Business requirements will in the case of a software solution, eventually be refined into software requirements.
- Software Requirements: A software requirement is a condition or capability of software, needed by a user to solve a problem or achieve an objective.
    - Functional Requirements: A Functional Requirement is a Software Requirement describing what the software should do.
    - Non-Functional Requirements: A Non-Functional Requirement is a requirement describing the quality attribute of that Functional Requirement. In contrast to the 'what' (described in the functional requirement), these are often referred to as the 'how' of requirements, describing how requirements are carried out.
        - Constraint: A constraint is a Non-Functional Requirement describing mandatory quality attributes.
        - A Property/Value Pair: A property/Value pair is a combination of two coupled values, used to describe the values of non functional quality attributes. When e.g. a constraint mentions a maximum execution time of 10ms, the property is speed, and the value is 10ms. The PropertyType and PredefinedValue make it possible to use a standard set of quality attributes with predefined values (When for example company standards or law prescribe this).

Relationships

- RequirementsRelationship: A requirementRelationship is a relation between requirements. To carry enough semantics, a number of specializations are derived from this relationship. The requirementRelation is introduced to capture the general information, which form the relationship (e.g. source and target). All the specializations either have extra attributes to capture more

information (e.g. Types) or have different source or target elements. The requirementRelationship element itself only has one attribute, describing the relation. The following specialisations exist:

- Dependency relation: A dependency relation is a relation indicating a specified dependency-type relation between two requirements. The following dependency-type are defined:
  - Requires: Requires is a type of dependency relation between two requirements, indicating that one requirement is needed to fulfil the other.
  - Excludes: Excludes is a type of dependency relation between two requirements, indicating that if one requirement is chosen, the other requirement in the relation can't be chosen.
  - Depends: Depends is a type of dependency relation between to requirements, indicating a general dependency that can not be captured by the above.
- Refine relation: A refine relation is a relation between two versions of requirements. When a refine relation is defined, all the relations to the replaced requirements are added to the replacing requirement. A requirement can have only one refine relationship, and can also only be defined between the same types of requirement.
- Influence relation: A influence relation is a relation between two requirements that have a cost or priority influence to each other. It captures the intuitions or experience that a requirement designer has when designing requirements. Only when it is known how a requirement is satisfied, the cost or priority relation can be established formally.
- Decomposition: A decomposition relationship is a relation between a part and the whole. The whole is an abstraction of its parts. It is used to build hierarchies in requirements (e.g. tree shaped). A type is used to indicate the fulfilment criteria of parts related to the whole.
  - AND: AND is a type of decomposition relation between two requirements in which the part is need to be fulfilled the whole. The parts actually add detail (e.g. an addition) to the whole, which needs to be fulfilled.
  - OR: OR is a type of decomposition relation between two requirements in which the part need not be fulfilled in order to fulfil the whole. It is used to indicate a possible alternative for fulfilling the whole.
  - XOR: XOR is a type of decomposition relation between two requirements in which exactly one of the XOR related parts needs to be implemented on order to fulfil the whole. It is used to indicate that exactly one of the XOR parts can be chosen for implementation.
- Affects: An affects relationship is a relation between a Functional Requirement and a Non-Functional Requirement. A Non-Functional Requirement will add quality attributes (or a mandatory constraint) to a Functional requirement.

2.3.6 Difficulties in the requirement metamodel and possible extensions

When deriving a common requirement metamodel based on literature study, one always has to make compromises. For the sake of simplicity, and also for reasons of focussing on the structure of requirements, some elements have been left out. In this section we briefly describe the main problems and controversies surrounding the model.

Default entry point

Almost all relevant requirement metamodels use a common entry point to access the requirement specification. Usually this is realised in the form of a common term such as document [Marshall'03], or Catalog [Vicente'07]. These concepts are created because these models are aimed at reuse. This is, next to simplicity, the main reason for leaving this concept out of our metamodel. A main entry point to access the requirement specification is however still needed. This can also be realized by defining one main requirement as the entry point, and define all other requirements as sub requirements of these requirements. In this way still a main entry point is defined, without the need for special elements for it.

Taxonomy

One problem with defining a taxonomy that is consistent with current literature and practice is that there is no consensus. What we have done is select a taxonomy that is often seen in literature, and seem relevant with respect to the structure of a requirement model. We must however admit that there is no real wrong or right in this respect. When a software project doesn't describe business requirements, but starts with top level software requirements, this doesn't mean that they use a faulty requirement model. What we tried to do in our common requirement metamodel is to enable most of

the requirement models to be captured in the structure we presented. This means making choices based on the discussed literature.

Decomposition

Another difficult aspect of specifying a requirement metamodel is choosing the relations between requirements. One specific relation, the decomposition relation is especially problematic, because of the many interpretations that can be given to it. Decomposition is also a very loaded term in the software engineering community, because it can be seen as the basis for every activity in software engineering; subdividing components in smaller sub parts and treat these independently. It is really the famous concept of divide and conquer or separation of concern. We have in our model, given interpretations to this concept, and added other types of decomposition found in literature. This resulted in the decomposition as presented above. Using this structure it should be possible to model most of the requirements models found in practice.

One difficulty and limitation to the decomposition relation that we must mention is the modelling of a requirement like: “3 of the following 5 sub requirements need to be implemented”. We recognise that this may be inconvenient to model using the presented requirement model. One option would be to separate all the options in which the 3 of the 5 sub requirements can be combined, and relate these groups with an XOR relation. Even though this is very inefficient, it is possible. We however expect that these circumstances will not arise much in practice, and we consciously choose not to change the model for this specific case. If the need for such a relation arises from actual practical requirement models, we could always adjust the requirement model with a specialized relation to enable it.

One last remark concerning the decomposition relation in particular is that it is an often studied relation it other software disciplines. One of these studies by [Keet’06] discusses the part/whole relation, and discovers eight different types of decomposition which are possible. How these kinds of decompositions relate to requirement modelling in particular is unknown, and needs further investigation before statements about this can be made.

Connection with other models; the conceptual model

We have chosen not to include a conceptual model, like seen in the work by [Marshall’03] into our requirements model. The reason for this is that it is very difficult to make a general intermediate (conceptual) model, to capture the information that is present in any requirement model that may exist. To be able to do this, one need to be as general as possible and the danger exists that one tries to ‘model the world’, We chose instead to present a model which should provide enough capabilities to capture most of the requirement models that exist, and use this as a basis. The relation to other requirement documents is left open for now.

Extra level relationships

We already stated that the focus in the presented common requirement metamodel lies in capturing structure of requirements. With this objective in mind the elements of the model have been selected. This is also why only a subset of the relations found in literature has been implemented in this model. To achieve the broader goal of this thesis to improve consistency between requirements and lower level design artefacts (architecture), we need to provide more relations aimed at traceability. These relations will be defined in a trace model, described in later chapters.

Refinement and Decomposition

One aspect of our discussion on requirement relations that quickly causes confusion is the similarities between the refinement and decomposition relation. We acknowledge that our usage of the relations is just our choice, and other choices could also be valid. We have tried to, based upon three research papers, investigated the usage of these relations and integrated them into one common set of relations that don’t overlap. We have specifically chosen to use the decomposition/part whole relation for adding details to a requirement. We have offered a way to model the fulfilment criteria to the composite. This enables us to build complex structures of requirements, adding detail to a requirement, and at the same time specifying when a requirement is fulfilled. The papers we discussed

in this chapter often used the term refinement for this. But after investigating the meaning of this relation we conclude that it can fully captured by decomposition.

We will only use the refinement relation in cases when a requirement is changed to another version. This situation will eventually occur in practice, and can't be ignored. We have reserved the (until now unused term) refinement to specify this kind of relation. This relation is very limited, and we clearly state that no two requirements can have the same source requirement as result of refinement. A requirement can only have one refinement. We included this relation for completeness of our discussion, but are fully aware that this is an area of research which is needs more research. This falls outside the scope of this thesis. In order to make more powerful statements about versioning and/or evolution in the case of requirements management, more research is needed.

Goal orientation:

One important concept in requirement engineering that we haven't touched yet is the concept of Goal orientation or Goal Oriented Requirements Engineering (GORE). In GORE, goals are used for eliciting, elaborating, structuring, analyzing, negotiating, documenting a modifying requirements [Lamsweerde'04]. In GORE, the most prominent element is a goal. A goal is a prescriptive statement of intent, whose satisfaction requires the cooperation of agents [Lamsweerde'04]. Agents are so called active components, and model the entity responsible to achieve a goal. An agent does not necessarily need to reside inside the software domain, but can also reside inside the environment or business domain of the goal. Goals are furthermore modelled in so called goal models. These models are being built using a AND/OR structures that describe how a goal is refined or abstracted into other goals. AND-refinement relates a goal to a set of sub goals, which states that satisfying all sub goals in the refinement is a sufficient condition for satisfying the goal. OR-refinement relate a goal to an alternative set of refinement which states that satisfying one of the refinements is a sufficient condition for satisfying the goal [Lamsweerde'03]. In goal models, goal refinement stops when every sub goal is realizable by some individual agent [Lamsweerde'03]. These goal models refine a high level strategic objective into lower level fine-grained technical prescriptions that can be assigned as a responsibility of a single agent [Lamsweerde'04]. When these agents reside inside the software domain these are usually seen as requirements on future software. When an agent resides inside the environment domain, it is usually seen as an expectation or assumption on the domain. GORE furthermore distinguished two types of goals, being functional and quality attributes. The functional goals are said to describe some set of desired scenarios, established in a clear cut sense. Quality goals however describe the preferred behaviour of those functional goals, often not to be established in a clear cut sense [Lamsweerde'04]. The quality goals are often used as an selection mechanism for selecting (sub) goals and imposing constraint on goal operationalization.

We will not describe every detail of the GORE approach in this chapter, as it is outside the scope of our research. We have therefore kept our discussion on GORE deliberately brief as it is not our focus to support this concept. It is however an important concept to recognise and we therefore briefly mention it for completeness. It is our belief that even with within GORE, our requirements model can still be used to map these concepts. Mainly because it uses decomposition mechanism is its main structure to model their so co called goal models. Using AND/OR structures they build up their models in order to refine and abstract their goals. These mechanisms are also available in our metamodel. The GORE Goal concept can furthermore be mapped on either a business or software requirement, depending on the level of refinement in the goal model. The higher level goals will usually be mapped upon the business requirement in our metamodel, and the more detailed, lower level goals will be mapped on software requirements. Being not custom tailored at supporting GORE, our model will eventually not fit every detail of the approach. The general concepts however seem to match to a certain extend, so that a general support of this approach should be possible.

2.4 Summary

In this chapter we gave an introduction into important concepts of requirement modelling. We described the importance of capturing requirements, and especially the importance of requirement models. Based upon three independent literature sources covering existing requirement metamodels, we have derived a common requirement metamodel. We have described the important elements of the tree literature sources contributing to our common metamodel.

In order to select the appropriate elements to include into our requirement metamodel, we have defined criteria for selecting these model elements. This criterion is aimed towards capturing the structure of the requirements, and we explicitly exclude any elements not directly related to the requirements phase. We consequently discussed the elements that are included in our metamodel, based upon the discussed literature sources. We discussed the basic requirement entities, taxonomies and relations to be used.

Having discussed the elements to include into our requirement metamodel, we presented our converging common requirement metamodel. This metamodel is later on used as a basis for consistency checking between requirements and design artefacts.

Lastly we identified the problems when composing a common requirement metamodel out of different literature sources. We have discussed the compromises that have to be made in order to achieve this.

3. Architectural design modelling

This chapter will discuss the current practice in architectural modelling. First, we will give an overview of what architecture is in Section 3.1. This will lead to a definition on architecture which we will use in this thesis. We will furthermore discuss the importance of having an architecture, how one can arrive at a good architecture, the importance of architectural views, and give an overview on the most used architectural development methods. Section 3.2 discusses Architectural Description Languages and a classification and comparison framework for them. This comparison framework will form the basis for a common architectural metamodel which we will discuss in Section 3.3.

3.1 What is Architecture

Architecture is a term which is often surrounded by a lot of misunderstanding and vagueness, not only in the software development world. The term architecture is actually used quite a lot in other practical domains, far away from software. One of the best known domains where the term is seen often is the building industry, in which an architect is a person who people can relate to; It is the person who designs physical buildings built with bricks and mortar. One big commonality between a software architect and a buildings architect is that they both need to give structure to something. The problem is however, that this 'something' is a very vague concept, and is easier to visualise in the building world, than in the software world.

When we look at the literature surrounding software architecture in particular, we find an overwhelming amount of people who tried to capture the pure essence of the term software architecture. We will mention a couple of definitions which at least have common elements in the definition:

- *"The logical and physical structure of a system, forged by all the strategic and tactical design decisions applied during development Booch (1991)"*
- *"The structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time (Garlan 2005)"*
- *"The organizational structure of a system or component (IEEE 610-1990)"*
- *"Software architecture is the fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution (IEEE 1471-2000)"*

We can see from these definitions above that a common keyword in all of them is (again) the word 'structure'. Other common terms used in these definitions are elements and the connections between these elements (even though different terms are sometimes used). The challenge is to combine these key concepts into a common working definition to use in this thesis. It will be this definition which will guide us in the construction of a common architectural model.

One influential publication in the area of software architectures in particular is the book by Lenn Bass, Paul Clements and Kazman [Bass'03]. This book identifies the lack of consensus on the definition of the term architecture. They treat a number of definitions, and give reasons for not accepting a definition as a proper definition for software architecture. We will quickly give the main criteria for accepting or rejecting elements of a proper architecture definition;

Modelling of element interaction

The main purpose of the architectural elements is to be used as an abstraction, focussing on the interactions and/or relations with other elements. This means that details of the element should be focussed on the interfaces that an element has with other elements, and keep other details hidden. Internal implementation details of elements should be hidden in an architectural view.

More than one structure

One very important factor which should be resembled in the definition of an architecture is that there is no such thing as 'the architecture'. Architecture often comprises of multiple structures, all describing different aspects of the architecture. As a result, the elements and relationships between the elements can have many different meanings depending on the particular structure it describes. This is exactly why a definition of architecture should not be restricted to one particular structure among elements. Both a description of the runtime behaviour of a system, and a partitioning into implementation units can be considered architecture. As a result, one can never 100% state what the elements and relationships of an architecture should be; this just depends on the kind of structure an architecture describes. We will come back to this aspect of architectural modelling when describing architectural views.

Every computing system with software has a software architecture

Basically every system has an architecture. The problem is that people don't often think this way, mainly because it isn't documented or written down. It is possible that a architecture is embedded in the implementation or even in the creator's head, and not made explicit in any model. We cannot say that such a system doesn't have an architecture. It must have some structure to it, even by the most basic interpretations. A definition of architecture should therefore not imply that an architecture only exists when it is documented or modelled in a certain way.

Behaviour is important

A description of the elements of an architecture through the use of a simple box and lines diagram is often meaningless and not considered architecture (or a architectural description). Often we see that elements only have a name hinting at the meaning of such an element. This problem is that this is very dependent on the reader of the architecture model. To have an architecture that has the same meaning for every stakeholder in the architectural process, the meaning of an element should be unambiguous. It is important that the (external) properties of elements should be described. At least their influences to other elements in order to interact with it, or the influences it has on the architecture as a whole, should be captured.

According to the aspects that we just described, the abovementioned definitions of architecture are not really suitable. One important problem with all of them is that they don't explicitly mention that an architecture has multiple structures describing different behaviour. Sometimes the definitions even have multiple flaws when we look at them from the perspective of the abovementioned criteria. The definition of Garlan, for example, also includes the process aspects that are included in shaping the architecture. It is highly questionable whether capturing information about the process adds to the understanding of the structure of a software system/component.

Having established that most existing definitions in the literature don't really fit all the criteria for a proper architecture, we have to come up with another one. Lenn Bass [Bass'03] comes up with a common definition which resembles most of the elements found in other definitions and still fulfils the criteria mentioned above. We will follow this definition as stated by [Bass'03]:

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

3.1.1 The importance of architecture

Having described what architecture is, we now focus on the importance of capturing an architecture in software development. We will discuss the important aspects of an architecture inspired by the work of [Bass'03]. Bass divided the important aspects of an architecture into three categories which we will discuss below:

Enable communication among stakeholders

The architecture of a software system can primarily serve as a common language or abstraction in order to describe the different structures of a system. In other words, it provides a vehicle for stakeholder communication [Bass'03]. What this means is that the architecture provides a common language or abstraction in which all stakeholders can discuss their concerns. In case of negotiation, it can provide a high level abstraction which enable different stakeholders with different interests (cost, reliability, implementation, etc) to communicate. Only when the architecture is made explicit, an architect is able to efficiently communicate with all the relevant stakeholders.

Represent early design decisions

Another important aspect of an architecture is to represent early design decisions. These decisions can be taken on a high level without having to physically implement the system. This is especially important because these early design decisions are often the most important to get right. Problems or faults in these early design decisions are most often the most difficult and costly to correct. Because they lie on the basis of every design step taken hereafter. Correcting problem herein will cause a lot of rework of other designs/implementations, based on these early decisions. The architecture furthermore makes it possible to take early design decisions and reason about it on a high abstraction level.

Defines constraints on the implementation

The architecture of a system offers constraints on an implementation, in the fact that an implementation needs to conform to the structural decisions of the architecture. By limiting a software implementer in the creative implementation process, this looks like a downside of an architecture. This doesn't necessarily need to be true. It offers a possibility to use separation of concern in order to assign separate chunks of work to specialists. Software implementers can now be assigned to implementation specific elements in their area of expertise. Software implementers hereby do not need tot be experts on all elements present in the architecture. The architect in turn doesn't need to be an expert on the implementation of every element, allowing him to focus on making the right architectural tradeoffs.

Dictates the work breakdown structure

The high level decomposition found in the architecture can be used as a guideline for the so called work breakdown structure. The work breakdown structure is dictates a number of things like planning, budgeting, inter team communication, file organisation, integration tests etc.

Enables specification of quality attributes and system quality

Quality attributes of a system are highly related to architecture because attributes need to be designed into a system from the start of a development trajectory. Only when quality attributes are designed into a system, they will have a high chance of being fulfilled. When quality attributes are important in the system design, then architecture is inherently also an important artefact.

Related to this, the architecture also enables the analysis of a system's quality properties before its implementation. Methods to enable analysis of this kind exist. Theyhelp to choose the right architecture for a system, based on the predictions of the delivered quality.

Enable reasoning and management of change

Change, as mentioned in previous chapter, sometimes looks like the only constant in software development. To be able to manage the changes that a system will undergo, the architecture is a very important element. It is one of the reasons why this thesis is investigating architecture in the first place. The architecture is a very important element in reasoning about change. Especially when we look at the implications of changes to artefacts further downstream the development process. In order to asses the implications of these changes, insight into the system elements and the relationships amongst these elements are needed. The architecture provides this information and allows for the reasoning and management of these changes. This enables a software project to asses the risk of changes, as soon as possible.

Prototyping

One last proof of early design decisions is the possibility to create early prototypes based on the architecture that has been created. The use of prototypes makes a system executable early in the development process. This does not mean that the product is fully functional, but it offers a look and feel of the product, in early phases. In this way, parts of the system can gradually be replaces by real implementations, and gradually improving the system. It also offers the possibility to evaluate the performance of a system as early as possible.

Offer a transferable, reusable abstraction of a system

As stated in [Bass'03], the benefits of reuse are at its greatest when it is carried out as early as possible in the development process. Reuse at the architectural level can be of great value when applied to systems with similar requirements. In architecture centred development, even requirements should be reusable. Also the other direction of the software development life cycle can be taken, in which the architectural reuse implies the re use of elements in the development steps after the architectural level (down to code level). This is also seen in disciplines like component based development. The aim is to start reusing as early as possible in the development process.

Reuse in Product lines

Software product lines are a specific kind of software system that can highly benefit from reuse at architectural level. Basically reuse is built into this kind of software. A product line is a set of software products that share a common set of software features that are reused across the set. The only way for this to be affective is that the architecture of this product line is designed with reuse in mind. Only then, different software product can be built using these shared features. The architecture should be designed with the needs of the entire family. It is clear that architecture is the core element in this approach. Without architecture, this approach would not be able to exist.

Use of externally developed elements

Reuse of so called Common Of The Shelve (COTS) software components is a trend we have seen in software development for some time now. These components are often integrated in an existing or newly built system. To be able to integrate these components, the components need to be interchangeable. The only way to make this possible is to have interchangeable components designed up front by an architecture. The architecture makes the use of external or COTS elements possible

Restrict the possible design alternatives

Patterns, which are blueprints to solve predefined problems, are becoming more and more popular. One of the reasons for this is the way in which they minimize complexity. They restriction the number of design choices in software cooperation and interaction, by offering standardized solutions. These limited set of (architectural) patterns can be combined into an architecture, limiting the selection that an architect needs to make.

Template based development

Template based development uses templates to capture the interaction and structure of elements. Templates are pieces of pre developed architecture that are written once and reused multiple times.

The basis for training

Lastly, the architecture can serve as a first introduction to new users. The architecture can explain how the new system is going to achieve its goals through the interaction of the elements in its architecture.

3.1.2 How to arrive at an architecture, and what to do with it

As mentioned before, architecture doesn't just exist; it is designed by an architect, who in turn is influenced by a number of stakeholders each with their own interest in the system. Stakeholders play an important role in the design phase of the architecture. The stakeholders express their concerns and goals with respect to the software system. These concerns and goals, are often expressed in business requirements and need to be translated into software requirements. This is the job of a requirements engineer, but it can be done by the software architect.

When the requirements are known, the architect's job really starts. He is the person responsible for translating these (often contradicting) requirements into an architecture satisfying the requirements. This means making tradeoffs and communicating with the stakeholders to achieve an architecture which all stakeholders support. Stakeholders often do not specify everything a system is supposed to do, leaving the architect with blank spots in the software specification process. Filling these blank spots is a challenging task made even worse by the fact that, next to stakeholders, also other forces play a role. As an result, the difficultly for an architect often lies in the forces that arise from not only the different stakeholders but also other influential parties. (depicted in Figure 10 ,taken from a column by Bass et al [Bass'05]).

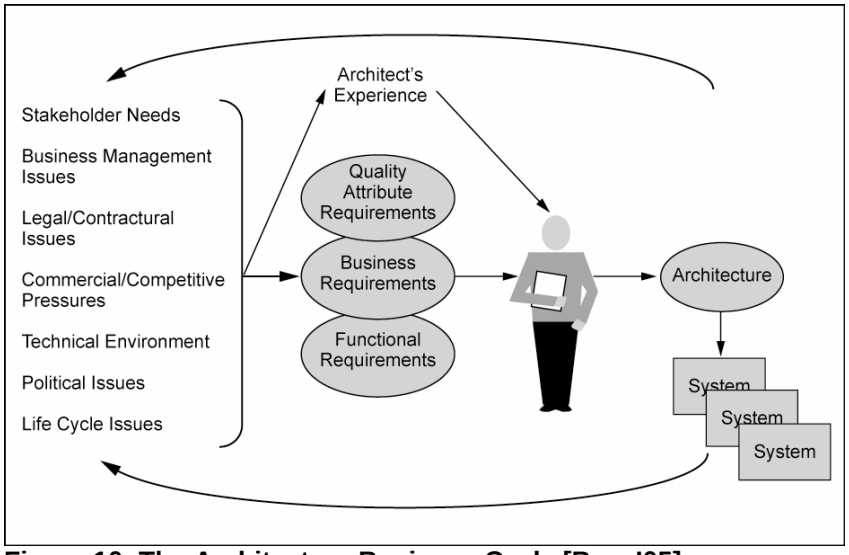


Figure 10: The Architecture Business Cycle [Bass'05]

The so called Architecture Business Cycle (ABC) cycle, depicts the influence of the forces on the architect when designing an architecture. It includes a feedback loop from the system and architecture to these influential forces, forming the (ABC) circle. These feedback mechanisms exist because the architecture and systems in turn influence the driving forces. We will give a couple of examples, taken from [Bass'03]:

- The architecture affects the structure of the developing organisation through the influence it has on the developing teams;
- The architecture can change the goals or markets that a company has because the system derived from the architecture enables new goals to be achieved, or new markets to be served;
- The system can give users a great benefit, which in turn can change the user's attitude to the system, and cause them to relax other requirements to enable this benefit.

Other feedback mechanisms exist, but it is outside the scope of this thesis to discuss these in detail. For further details about the ABC cycle, we refer to the book of [Bass'03].

Once an architecture is crafted to fulfil the requirements of the stakeholders and other driving forces, the architect needs to communicate the architecture. This implies documentation of the architecture. Every structure that is created needs to be documented in an appropriate way for each specific stakeholder to be most useful. These documents, tailored to describe a certain structure are called views of an architecture. We will discuss this element of architectural design in the next section.

Once the architectures are documented, a selection among candidate architectures needs to be made. Several different methods like ATAM (Architectural Trade off Analysis Method) and CBAM (Cost Benefit Analysis Method) have been developed in order to support this selection process. It is again outside the scope of this thesis to discuss these selection methods, but we mention them in order to give a clear meaning of the process in which architecture is developed and used.

The final step in which architecture plays a role is the implementation and maintenance phase. These phases are supported with the documentation of the architecture. It is important to keep consistency between the implementation of a product, and the architecture describing it.

3.1.3 Architectural views

Previously, we shortly touched the subject of having different structures to describe an architecture in order to satisfy the need of specific kinds of stakeholders. There is however more to say about this topic. One of influential work on the topic of architectural descriptions, and in particular generally accepted trends in this direction, is the IEEE Recommended Practice for Architectural Descriptions of Software Intensive Systems [IEEE1471'00]. In this document, the IEEE organisation tries to find a so called accepted framework for codifying architectural thinking. This means that they try to find a consistent meaning for terms used in architectural modelling according to current use in the industry. One general accepted notion behind this framework is that an architecture should always be modelled with the stakeholder in mind, in order to meet his concerns.

The core of this document is a conceptual model of architectural descriptions. This model shows the relationships between the different concepts in an architectural description, as seen in Figure 11.

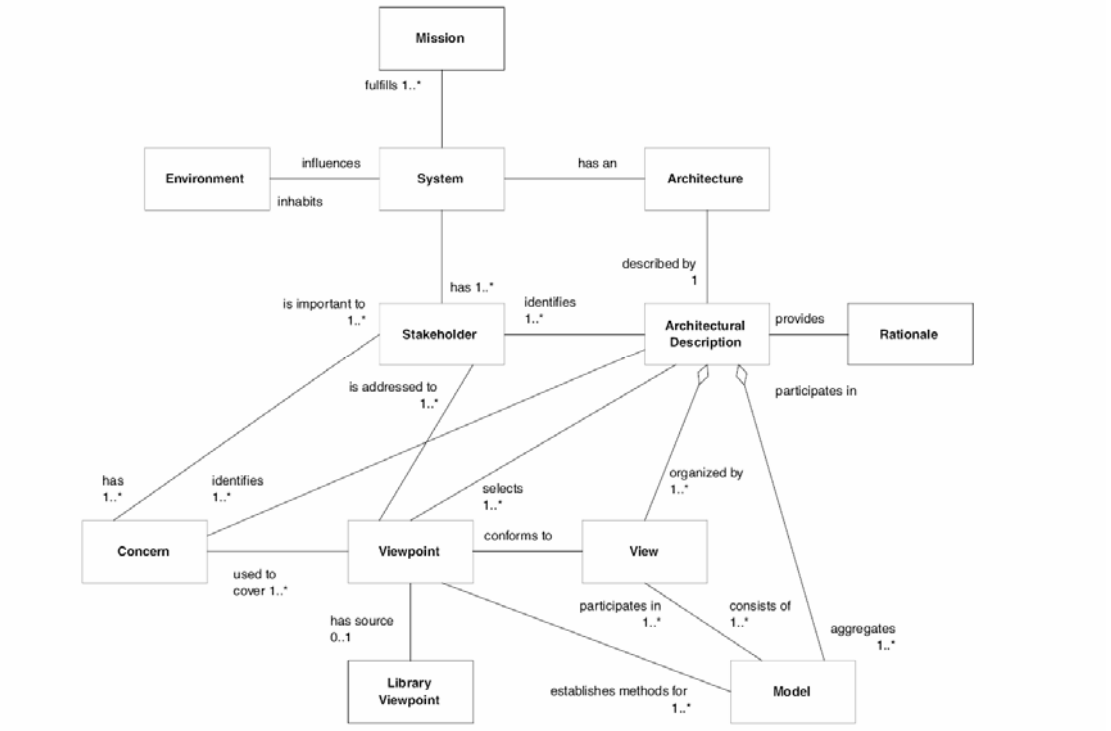


Figure 11: Conceptual model of architectural description [IEEE1471'00]

At the centre of the model the system itself is positioned. This system is defined as a collection of components organised to accomplish a function or functions. We see that the system inhabits a certain environment or context, influencing the system. This environment can have different meanings, like for example political forces, constraints on the development, or other systems with which the system needs to communicate. Furthermore, a system has a mission which describes its use or the reasons for its existence.

Next to this, the system also has stakeholders, as we have seen in the previous chapter. These stakeholders have a certain interest (or stake) in the system. To express the interest that a certain stakeholder has in the system, this information is captured in a so called concern. A concern is an interest that the stakeholder has in the development, operation or other aspects of the system. Examples of stakeholder concerns are; function, performance, evolution, structure etc.

The architecture of a system is a conceptual item, which can possibly only reside in an architects mind (as we have seen earlier). This approach recognises that every system has an architecture, whether or not described in a model or language. Such an architecture exists, but has little meaning to the outside world, except for the architect who designed the system. In order for an architecture to really be useful, this architecture needs to be communicated to the outside world. An architecture, is therefore described by a so called architectural description. This architectural description is a (collection of) physical documents or models describing the system. We need to stress that this is depicted in the conceptual model as a single model element. This does not mean that an architecture can be described entirely by only one model or document. What we see instead is that an Architectural Description (AD) consists of a number of views and viewpoints. It is these views and viewpoints in which we are most interested in this section.

Views, as seen in the conceptual model, represent the system from the perspective of a particular concern, in order to meet the needs of certain stakeholders. We could say that these views describe the different structures that comprise an architecture as defined in our definition; Every structure is described by one view. The conceptual model also mentions the use of viewpoints. Viewpoints are basically blueprints or templates for certain views. A viewpoint describes how to construct a certain view in order to satisfy a certain concern. It describes the purpose, audience and techniques for creating a particular view. These techniques or methods for creating the views are described in the models associated to a view.

Viewpoints and views are in a similar way related as classes and object are in the object oriented world. An architecture has multiple viewpoints describing the different interests that stakeholders can have. The actual description of such an interest is embedded in the views of that viewpoint. Viewpoints are highly related to the aforementioned multiple structures of which an architecture is comprised. Each architectural structure has a related viewpoint. It is the recommendation of the IEEE organisation to have at least every concern addressed by one viewpoint. Furthermore, also a rationale for including the viewpoints needs to be included. This rationale is meant to capture evidence that alternative architectural concepts where considered during the selection of a viewpoint. The IEEE organisation does, in the 1471-recommendation, not prescribe a particular set of views or viewpoints to use. There are however numerous frameworks which describe a set of standard views to use in order to cover the architectural needs of most of the stakeholders. We will discuss two of these frameworks shortly below.

The four plus one view

Philippe B. Kruchten, in 1995, published an influential paper on the topic of using 5 concurrent views on one architecture to properly describe the architecture [Kruchten'95]. The so called 4+1 view model was designed to offer a framework of standard viewpoints, each describing a set of specific concerns. The main reason behind the framework is that an architecture can hardly be described entirely by one diagram. In order to address this problem, the author introduces five different views. Figure 12 depicts the five views introduced by Kruchten together with its intended users. In concordance with the abovementioned conceptual model of architectural descriptions we have to note that the views mentioned by Kruchten must more be seen as viewpoints than actual views. The views by Kruchten describe a certain perspective on the architecture that needs to be addressed. The actual documents or models which describe the 5 different viewpoints could be seen as the real views.

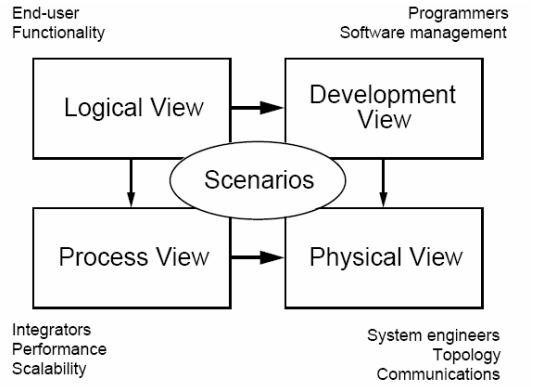


Figure 12: The 4+1 view model

The author of the 4+1 view model defines a set of element to use in each of the views using components, connectors and containers. This is inspired by the work of Perry & Wolf, who introduced a formula for describing software architecture as; Software architecture = {Elements, Forms, Rationale}. Next to these elements, they also define elements to: capture the forms and patterns that work; capture the rationale and constraints; connect the architecture to some of the requirements. [Kruchten'95]. To guide the users of the 4+1 view model in using the particular views, the authors have defined how the components, connectors and containers should look like for that particular view.

We will shortly discuss the 5 viewpoints that Kruchten mentions in his work:

- The logical view: This view describes the structure of the system, which is highly dependent on its design method. When Object Oriented methods are used, this could include class diagrams. It could however also include an Entity Relation diagram, when conventional procedural methods are used. It is mainly focussed on the functional requirements and services that system should deliver. Often decomposition of a system into smaller abstractions is performed in this view. The author suggests the use of the Booch notation for this view.
- The process view: This view mainly focuses on the Non-Functional Requirements. It is used to model things like performance, availability etc. In the original work by Kruchten, examples of things that are addressed are concurrency, distribution, integrity and fault tolerance. Also logical chunks, identified in the logical view, are assigned to threads of control. Processed are defines as a group of tasks that form an executable unit [Kruchten'95]. This view partitions these processes into smaller tasks that can be scheduled on hardware. The system can in this way be described on multiple abstraction levels. The suggested notation for this view is again based on the work by Booch (for Ada tasking). Multiple architectural styles can be used according to the author. Suggestions include pipes and filters, client server etc.
- The physical view: The physical view is concerned with the mapping of software elements into the hardware components. This viewpoint captures the distributed aspects of a system over its hardware components. It is often concerned with the Non-Functional Requirements reliability or performance. Other aspects of this view are the stimuli and responses of the system. The suggested notation that the author mentions is a rather infamous notation called UNAS which is not further explored.
- The development view: This view describes the organisation of the software modules in the development environment. This view splits the system into smaller easily manageable chunks to assign to the development teams. By splitting the software into subsystems to be separately developed it support the work and cost planning. This view addresses the so called internal requirements related to the ease of development, the software management, reuse, the constraints offered by toolsets or cost limitations. The suggested notation is a limited form of the Booch notation, and the suggested style is the layered one. The author suggests four to six layers of subsystems.
- The last view, the scenario view is a different kind of view. It is actually an illustration of the other views using scenarios of the system. These scenarios are used to illustrate the other views, and thus prove their usefulness. The architecture should partially be evolved from these scenarios, and should be very indicative for that system. This scenario view could also be seen as a kind of consistency check for other views. If a scenario can successfully be modelled without

contradicting one of the views, this hints at consistency amongst them. The scenarios are expressed using object scenario and object interaction diagrams.

Recommendations in connection amongst views and the process to use

The arrows in Figure 12 correspond to connections that exist between views, indicating their dependence on each other. The author gives a number of design rules and heuristics to arrive at a coherent and consistent set of views. He also gives suggestions on the process to use when developing a system using the 4+1 view model. The author suggests using an iterative process, in which a scenario based approach, is preferred. It is outside the scope of this thesis to discuss this in detail and we refer the interested reader to the work by Kruchten [Kruchten'95]. It is interesting to note that the four Plus One approach forms the underlying theory for the development process Rational Unified Process (RUP).

Tailoring

The 4+1 View model is meant to be adapted to the particular needs of an architect in order to satisfy the stakeholder concerns. It needs to be tailored to the development process and techniques used. This means that it should be used as a kind of guideline and not as a rigid textbook to use in architectural modelling. Scenarios are advisable to use in every circumstance, but the other views (or viewpoints) should be tailored to the needs of the stakeholders and its concerns.

Viewpoints defined by Bass

Another influential source describing a set of viewpoints to use in a development process is defined by Bass [Bass'03]. He describes a categorisation of views to use in architectural modelling. Three types of views on an architecture are defined, based on the nature of the elements that they contain. We will discuss these three viewpoints in a point by point manner. We will also relate these to the 4+1 view model of Kruchten, as these viewpoints are highly related.

- Module structures
  In this view, the elements that are used are modules representing units of implementation. The modules are assigned areas of functional responsibility. The emphasis is on the function of the module and its interaction with other modules. This kind of view can be compared to the logical view in the 4+1 view model. The module view includes the following commonly used software structures:
  - Decomposition: This kind of structure is often seen in order to reduce the complexity of the modules into smaller subcomponents. The relation between the modules is often called "is a sub module of". It is very natural for an architect to work in this way. Starting with a small set of elements that describe the general function and decomposing those into smaller more specific subcomponents with added detail of functionality. Decomposition also has other advantages such as improved change management (limiting the scope of change), and it can form the basis for work assignment, documentation etc.
  - Uses: When a module uses the functional behaviour of another element in order to function correctly, a uses relation can be established between the two modules. One big advantage of this structure is the ability to use incremental development of components by incrementally adding functionality through the addition of components. In this way, the system can become (partially) operative early in the development, not delaying the delivery until late in the development.
  - Layered: A special case of the uses relation is the layered relation. Here, one layer can only communicate (or use) the services of the directly adjacent layer. Other uses relations are not allowed. This kind of structure is used as an abstraction mechanism in which one layer hides details of underlying layers to the layer above.
  - Class or Generalisation: This structure is used to generalise modules, called classes in this case. Classes with similar behaviour or capability are captured into a common structure. The relations between these classes are either called 'inherits from' or 'is an instance of'. Advantages are reuse of similar behaviour and incremental addition of details (or versioning). This is an often seen structure in Object Orientation.
- Component and connector structures
  In this view, the elements are runtime components and connectors. This view is aimed at the executing components in a system and the interaction between these components. It is purely

based on the executing behaviour of the system such as data flows, shared data behaviour or possible parallel components. The relation between the processes is defined as attachment, describing how the components and connectors are hooked together [Bass'03]. This view can be compared to the process view in the 4+1 view model. The following software structures are used in this view:
  - Communicating processes: This structure couples two processes or threads together through communication between two processes. This structure is important to capture because it can be used to analyse the performance or availability of a system by analyzing the interaction between processes and the implication that this can have.
  - Concurrency: This structure is used to determine opportunities for parallelism (concurrent execution) amongst components. It is often used to indicate where possible contention points in a system can occur. The units in this view are components, and the connectors are logical threads (later mapped to physical threads).
  - Shared data: This structure is used to show how data is produced and consumed by runtime software elements. This view is often used to analyse the performance or data integrity of the persistent data.
  - Client Server: This structure is used to indicate the cooperation between a server and its clients with its protocols in between as connectors. The components here are either the server or the clients, and the connectors model the communication between the two as either a protocol or messages.
- Allocation structures
  This view is aimed at the interaction between the software elements and the environment. The environment can be a hardware system, a development team or physical files. The mapping of software elements to these environment elements gives insight in component allocation, assignment of work or the physical location of certain files. This view corresponds to both the development and physical view as mentioned in the 4+1 view model.
  The following structures are often seen in these views:
  - Deployment: Used for the assignment of software to hardware. Here the elements are software, hardware and communication pathways. The relation that is established here is referred to as 'allocated to', which describe what software elements reside on what physical hardware. This view is typically used for performance analysis, data integrity checking, availability management or security analysis.
  - Implementation: Used for the assignment of software elements to file structures. This is especially useful for the management activities surrounding the development.
  - Work assignment: Used for assignment of modules to development teams or individual people. This view is very useful for the earlier mentioned work breakdown structure.

3.1.4 Software development methods

In this section we will briefly discuss two modern software development methods, which include the creation of an architecture. We describe these methods in order to get an understanding of the steps involved in designing an architecture. These methods are not focussed at creating an architecture, but involve the entire development process. We discuss them in order to increases our understanding of how an architecture is created.

First we will discuss the Rational Unified Process (RUP), frequently used throughout the industry. Next we will discuss the Analysis Synthesis method. This is an interesting approach grounded in the problem solving theory, introduced in the previous chapter.

Rational Unified Process

The Rational Unified Process, first published by Jacobson [Jacobson'99], is a software development method which has in recent years become very popular in the software development industry. It is created by Rational Software, nowadays a division of IBM. RUP is meant to be a customizable framework which means that it can be adapted to the organisations needs. It is not meant as a rigid set of rules which need to be followed. Instead, the company can tailor the process to fit its own needs and adjust the complexity and time spent per phase as the company sees fit. RUP is an iterative, incremental, use case driven and architecture centric development process. These keywords really capture the essence of what RUP is all about. We will elaborate on these keywords in the following sections.

Iterative and incremental

RUP describes a development process which is referred to as an iterative and incremental process. This first term refers to the fact that the entire development process is centred on an iterative way of working. The well known water fall method in contrast has a fixed trajectory of development steps to be followed. Once a development phase is closed, it doesn't offer a way back. Instead RUP offers a process that can be restarted once the deployment phase has been reached. (Seen in Figure 13). After evaluating a project, possible improvements are identified and the process is restarted. RUP has a number of process disciplines which are recognisable in the figure. These process disciplines more or less resemble the well known development phases, also found in the waterfall model.

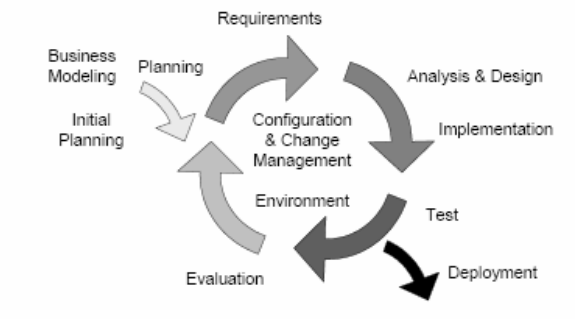


Figure 13: Iterations in the RUP [Noppen'07]

The main reason for using an iterative process is the incremental aspect of the method. The development process of a software product is divided into chunks of development, called iterations. Each iteration results in a release of the system with added value compared to the previous one. (Called an increment). We can graphically depict the process of in Figure 14.

We see that the total development time is divided into a number of phases: inception, elaboration, construction and transition. These phases refine the artefacts of a system. Each of these phases is in turn divided into iterations. Most of these iterations will cover all of the process disciplines. The inception phase however dies not. This can easily be explained by the fact that during the inception phase, more work needs to be done on business modelling and requirement solicitation instead of implementing software. We furthermore see that the amount of work per process discipline changes as the iterations move throughout the phases. From the elaboration phase on, we can see that the iterative nature is clearly noticeable. The amount of time spent per process disciple changes during the phases.

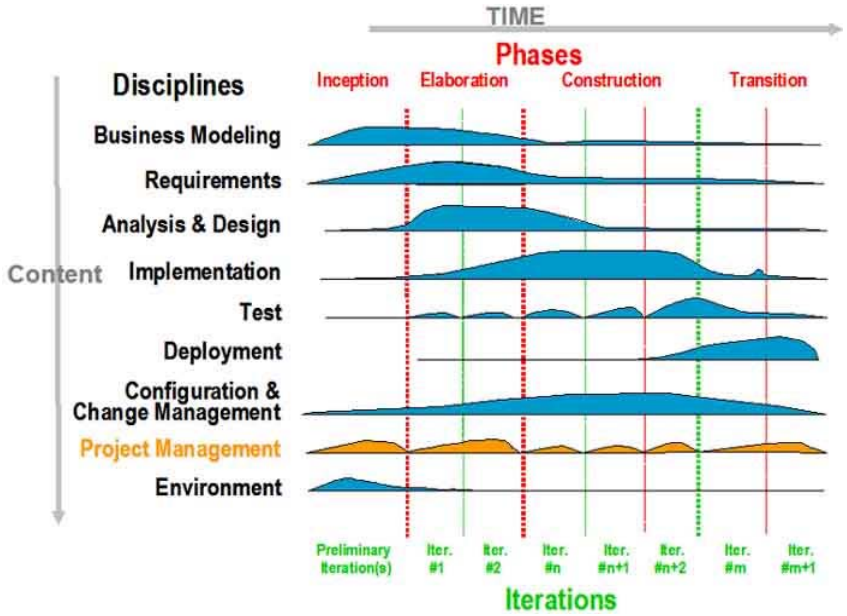


Figure 14: RUP Framework.

Use Case Driven

RUP uses use-cases to describe the functional requirement of a system. A use-case is defined as a sequence of actions that the system provides for actors [Jacobson'99]. Each iteration is driven by a set of these use-cases, and is aimed at implementing the functionality described in the use cases. After a certain iteration and increment is deployed, the functionality described in the use case should be available.

Architecture Centric

An important aspect of RUP is its focus on the architecture of a system. This means that the architecture is the key artefact in this method. RUP supports multiple architectural models or views to support the specification of the architecture. We can clearly see the importance of the architecture resembled in Figure 14. We see that a large amount of time is spent on the analysis and design of a system in the early phases of the development. This is resembled by the amount of work being carried out on business modelling, requirements and analysis and design disciplines in the early development phases (inception and elaboration).

Synthesis based Software Architecture Design method

Synthesis based software architecture design method (or Synbad), described in [Tekinerdogan'00] is a relatively new development method. As the name suggests it uses synthesis as the basis for this approach. Synthesis is a problem solving process that has proven its use in traditional engineering disciplines. The focus in synthesis is technical problem analysis that identifies problems and structures them into sub problems that are independently solved. These are later integrated into the overall solution [Tekinerdogan'00]. Solutions to these sub problems are found using so called domain analysis. A solution domain is typically a domain of expertise, from which solution concepts can be extracted. Alternatives to these solutions can be extracted, and evaluated in an explicit alternative space analysis.

This approach fits in the overall picture of problem solving, already described in the chapter covering requirements modelling. In the Synbad approach, the requirements are seen as the problems for which a solution needs to be found. These solutions are derived from the solution domains which contain stable concepts to use in the architectural modelling.

Figure 15 illustrates the Synbad design method in a graphical way. Here we see the different steps (and sub steps) in this approach. We also see that this approach uses iterative concepts, like seen in RUP, based on the plan descriptions in the figure. We can see that solution domain analysis can influence the requirement phase, causing the development to return to requirement analysis.

What we see from this schematic figure is that Synbad is focussed on arriving at the right architecture, and not at the development steps hereafter. This in contrast with RUP which also includes implementation and testing

It would go beyond the scope of this thesis to discuss every design step of the Synbad approach in detail. It is however clear that this approach fits the general problem solving process which we already described in the chapter covering requirement modelling very well. This approach is also used to design the architecture we will use to validate our architectural metamodel.

One interesting point when using this method, is whether a feedback loop from the architectural specification to the requirements analysis phase needs to be included in order to maintain consistency between the requirement specification and the architectural specification as seen in Figure 15. This would support the existence of methods or tools to support the maintenance of consistency between these two.

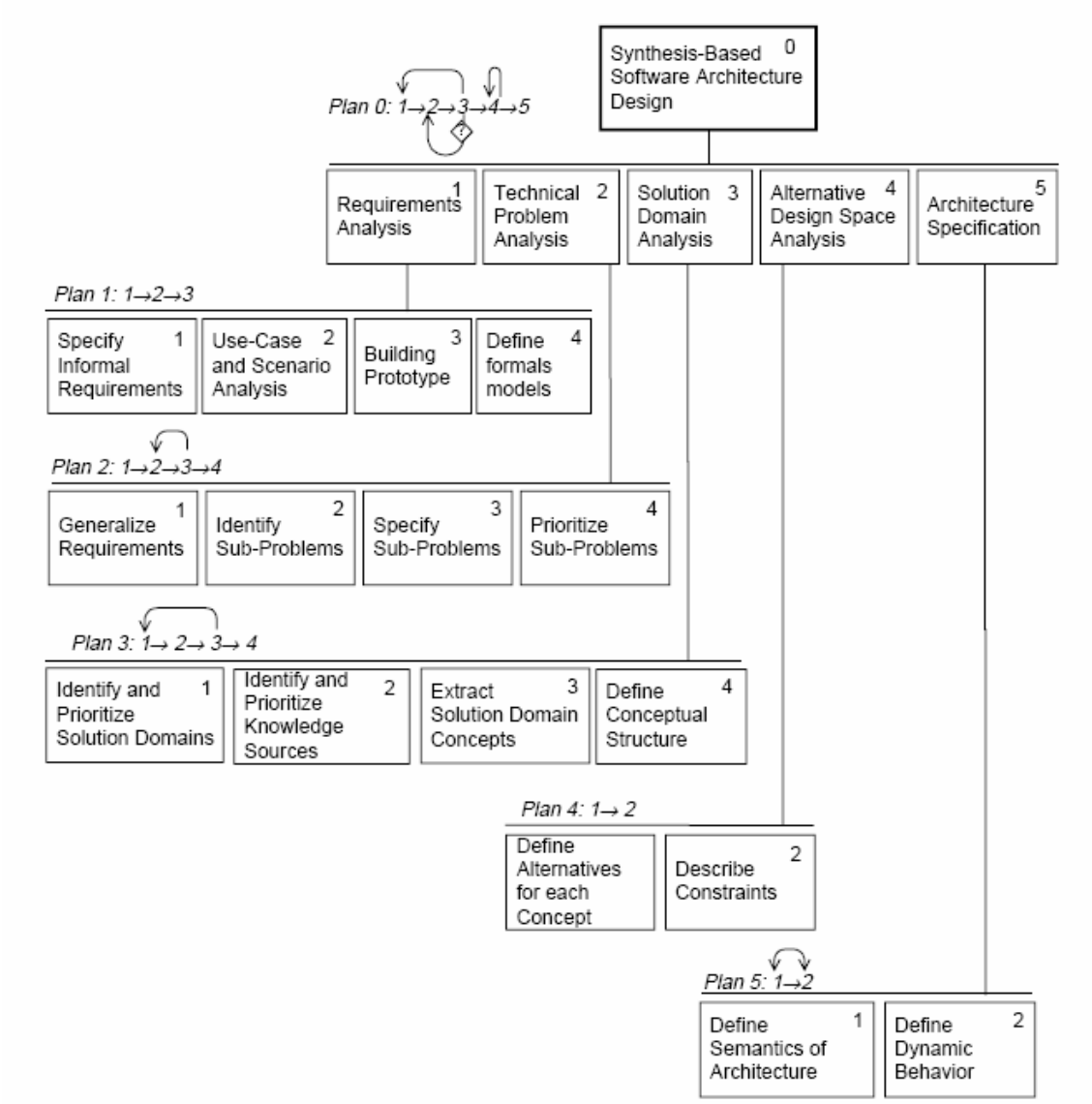


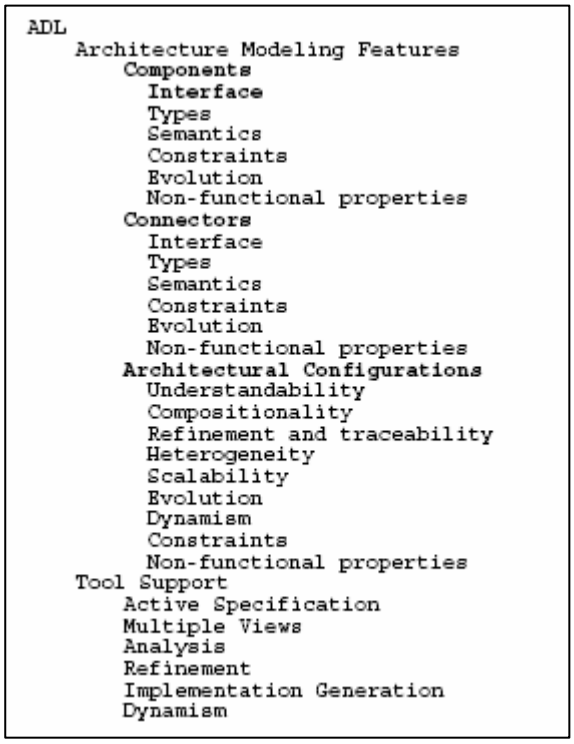
Figure 15: Synthesis based Software Architecture Design Approach [Tekinerdogan'00]

3.2 Architecture Description Languages

Architecture Description Languages (ADLs) are becoming increasingly popular as ways to model the architecture of software systems. It is also identified by the IEEE organisation as an innovation resulting from the increased attention in architectural level [IEEE1471'00]. This section will discuss what elements make an ADL useful and what elements should be modelled in an ADL. This discussion will be based on the influential work in this area, published by Medvidovic [Medvidovic'97]. In his work, Medvidovic presents a classification and comparison framework to compare different kinds of ADLs. A total of eleven ADL's are compared on the basis of this framework. The main reason for the existence of this comparison framework is the lack of consensus that exists in the software community about what an ADL should offer. The work by Medvidovic is one of the first to present a definition and relative concise classification framework for ADL's.

Medvidovic states that an ADL must explicitly model components, connectors, and their configurations. Furthermore to be truly useable and useful, it must provide tool support for architecture based development and evolution. These statements are the basis for the comparison framework. These tree elements are very much grounded in other literature surrounding architectural descriptions. One of them is the work by Kruchten which we discussed earlier. He also mentions that each view should be characterised by a set of elements to use, being; components, containers and connectors.

Next to this, also form, rationale, constraints and patterns-that-work should be captured. Kruchten based his work in turn on earlier works by Perry & Wolf. We can see a high resemblance with the work by Kruchten and the comparison framework, as depicted in Figure 16.



The three main elements: components, connectors and architectural configurations are clearly visible, in Figure 16 and therefore expressed in bold. Only when a notation has means for explicit specification of these components, can it be considered a real ADL.

Another explicit demand on an ADL is the modelling of the interfaces of components in an ADL. Only when those are specified any kind of information about the architecture can be inferred. The authors of the framework even state that without this vital information, a notation is nothing more than a collection of boxes and lines between them.

The other aspects of components, connectors and configurations, mentioned in the framework, are desirable but not essential for a notation to be classified as an ADL. The same is true for tool support in an ADL. Even though tool support of an ADL can render an ADL more useful and usable, it is not a necessity for an ADL.

We will discuss the essential elements of an ADL in the next section

Figure 16: ADL Classification and comparison framework

It is important to note that there are many kinds of ADLs, even as many as there are views on a certain architecture. This is very logical, because an ADL also only describes an architecture from a certain point of view. In this way, we could see an ADL as a kind of view which satisfies a certain viewpoint for the stakeholder. The ADL classification framework presented above is not dependent on any specific kind of ADL, describing a certain domain. Instead, it identifies the components that every ADL, independent of the domain, should have.

3.2.1 Components

A component of an ADL is defined as a unit of computation or a data store. It can be seen as an element that either caries data, or carries out some kind of computation. This can be on any scale, ranging from a single function or procedure to entire architectural components (E.g. a middleware component). As pointed out earlier, the interface of a component is one of the most important elements of the component, and should always be modelled in an ADL. Also other possible features of a component can be specified. We will elaborate on these shortly;

Interface: The interface specifies the possible interactions that a component can have with its outside world. It can be seen as a kind of contract that it has with the outside world. It specifies what services the component can deliver. The only thing that an outside component can see is the interface of a component. The interface of a system can be accessible through a number of ways. It can be simple messages that a component provides, complex functions with input and output variables, or sometimes simple shared variables. The interface of a component can also specify the services that a component needs from other components (uses relation). In this way other components can be developed in order to satisfy the needs of another component.

Type: Component types are basically mechanisms to enable components to be reused. They encapsulate functionality into reusable blocks [Medvidocic'97]. By providing a component with a type, a component can be instantiated multiple times in order for the functionality to be reused across

architectures. Another advantage of types is that they improve the understandability of an architecture, by creating recognisable properties of components, that can reoccur in the architecture.

Semantics: Semantics are the (high level) description of the behaviour of a component. To capture this information, a model of the semantics can be included. These models can be used to perform analysis, enforce architectural constraints, and ensure a consistent mapping of architecture from one level of abstraction to another [Medvidocic'97].

Constraints: A constraint on an architectural component is very similar to the constraints seen in requirements modelling. It is a property of the system which needs to be fulfilled. Otherwise the system will be unacceptable or less desirable. It is again a mandatory property which needs to be fulfilled. They are used to ensure the correct usage of a component, enforce the boundaries of usage and to establish dependencies amongst components.

Evolution: As mentioned earlier, change is almost a constant factor in software engineering. For this reason, change should also be reflected in architecture, and as a result in an ADL. In the comparison framework, evolution is defined as: the modification of (a subset of) a component's properties (E.g. interface, behaviour or implementation). ADLs should provide for mechanisms to ensure that evolution can be systematically applied to the architecture. An example of such a mechanism could be sub typing (or generalisation).

Non-Functional Properties: The Non-Functional properties of an architecture are a reflection of the Non-Functional Requirements. They should be explicitly modelled in an architecture in order to properly integrate them into a software system. It is especially useful for the enforcement of constraints, performing of analysis etc.

3.2.2 Connectors

We could say that connectors are the glue that holds the components together. They are used to model the interactions between the components and may contain information about the way this interaction takes place. Connectors need not be separate compilable units, like for example a CORBA component. It can also be simple things like shared variables, common keys in a relational database or a remote procedure call.

Interface: The interface of a connector specifies the interaction points that the connector has with components. The interface of the connector is a collection of interfaces defined by the components to which it is attached. A connector can in this way offer services, offered by connected components. A component never communicates with other components directly via its interfaces, but does this via connecting connector interfaces.

Types: Connector types are mostly useful for reasons of reuse, both across and within architectures. They are modelled as abstractions that encapsulate the component communication, coordination and mediation decisions [Medvidocic'97]. This is either done using a built in enumerated types or as an extensible type system.

Semantics: The semantics of a connector defines the high level behaviour of a connector. It mainly expresses the specifications of the interaction protocols that are used. This can offer advantages such as the ability to perform interaction analysis.

Constraints: The constraints are an important aspect of a connector, because they limit the way in which the interaction protocols are used. They are intended to ensure adherence to the interaction protocols of the connector. One could for example limit the time that the communication between two components may take, or even the amount of components that are allowed to communicate through the connector.

Evolution: Evolution of a component is more or less the same as evolution of a component. When certain adjustments to the (often complex) connectors of a system need to be made, it is desirable to have evolution support in an ADL. These changes can be changes on the interfaces, semantics or constraints of the connector. This is again often achieved through the use of sub typing.

Non Functional Properties: The non functional properties of a connector describe behaviour that is not described in the semantics of the system. They are properties which are important for a correct functioning of the connector. They can be used to enable simulation of runtime behaviour of a connector, analysis of the connector, enforcement of constraint or assist the selection of 3r party connectors [Medvidocic'97].

3.2.3 Architectural Configurations

The architectural configurations are basically the most important elements in an ADL. They describe the general structure of an architecture, using the components and connectors. It can be seen as a connected graph that is formed using components and connectors. Its main use is to ensure a correct combined functioning of the composite system. It also enables assessment of concurrent and distributed aspects of the architecture such as deadlocks and performance, reliability or security issues [Medvidocic'97].

The comparison framework also mentions a number of features that can be expected when modelling an architectural configuration in an ADL

Understandability: The understandability of the configuration relates to the role of the architecture as a communication vehicle across stakeholders. It must model information in a simple and understandable syntax. The structure of the system should be understandable from the configuration specification, without understanding all the details of the components and the connectors.

Compositionality: The compositionality of the architecture refers to the ability to model the system at different levels of detail. It offers the designer of an architectural configuration the ability to abstract away from details. In this way, entire (sub) architecture may be captured into a single component (to be used in architectures).

Refinement and Traceability: The refinement and traceability aspect of a language refers to the possibility to refine an architecture into an executable system and the possibility to trace changes to these refinements. This specifically relates to the automatic generation of executable code from architecture. It is often mentioned that the goal of an ADL is exactly this; Bridge the gap between informal "boxes and lines" diagrams and programming languages, which are deemed to low –level for application design activities [Medvidovic'97].

Heterogeneity: Heterogeneity of an ADL basically refers to the level in which an ADL is open to heterogeneous components and connectors to be used throughout the architecture. The heterogeneity reflects to the granularity of the components, or even on the language which is used to specify them.

Scalability: This aspect of a configuration speaks for itself and refers to the ability to cope with large and growing systems.

Evolution: Evolution refers to the ability to support evolution in architecture. This is important because evolution is often the most costly activity in software development. Evolution should be supported by ADLs on the level of component and connectors.

Dynamism: Dynamism is closely related to the foregoing, but is directed at modifying the architecture while it is executing. This requires special features by an ADL to support this. This is especially important in safety critical systems that cannot afford to go offline for changes on the architecture.

Constraints: Constraints in a configuration are supplemental to the constraints specified by the components and connectors. They are often derived from them or depend on them.

Non Functional properties: The properties describe the non functional properties at system level. These can for example be used for selection of appropriate components and connectors, enforcement of constraints.

3.3 Towards a common Architectural metamodel.

In this section we will describe a common architectural metamodel which we will use in order to serve as a metamodel for architectural modelling in general. This architectural metamodel will be a common denominator for architectural modelling, used to map different kind of architectural models onto. Once these mapping can be achieved, we can use this metamodel as a basis for establishing the relations between requirements and architecture. Before we discuss the architectural model we first have to come up with criteria for selecting the model elements, and deciding which modelling aspects to support. After this, we will introduce the architectural model and offer an explanation of its components.

3.3.1 Criteria for composing the common architectural model.

Modelling architecture can be done in a wide variety of ways. Many different views on architecture exist, each focussing on different aspects of the system. When composing a metamodel to capture the information of different architectural structures or views, we have to restrict ourselves. It is infeasible and also not desired to come up with a metamodel to model every architectural view. One reason for this is the simple fact that there is no standard in architectural views, so new views can come into existence every day. When an architect comes up with a new way to model some aspect of a system, this could be considered a new view. Another reason is that the metamodel would become to complex and unusable in practice. We therefore choose to restrict ourselves. We will first state our overall focus in creating our metamodel and hereafter state criteria restricting what the model should and should not support. We discuss these criteria in this section.

The common architectural metamodel must be based upon the common characteristic found in Architectural Description Languages: Components, Connectors and Configurations.

This will provide us with the structures to support the common denominator found in ADLs. This common denominator is given by the main elements of the Classification and comparison framework, discussed earlier. This will enable us to capture a wide variety of architectural structures.

Limited support for reuse

When creating a common architectural metamodel, we will not aim at the support for reuse. We excluded this support from the requirement metamodel, and we therefore see no need to explicitly model it in our architectural metamodel. Reuse is (as described earlier), often supported trough the use of typing of components. These types can be reused across different architectures. Reuse is often mentioned as the key element behind product lines and other techniques aimed at reusing a standard set of features across a set of products. We will not focus on any specific approach of software development, and therefore will not aim our model at explicit reuse mechanisms.

Limited support for evolution

Evolution of architecture is something we can not neglect to support in a metamodel. The main motivation for this is that software, and also its architecture, will almost always be affected by changes. These changes can be the result of changing requirements, changing environment of the system or possibly changes that the architect makes to achieve certain quality factors. The need for support of evolution is also recognised by Medvidovic [Medvidovic'97], by stating that an ADL is only useful when support for evolution is available.

We have seen in the ADL comparison framework that evolution of architecture is often supported by special constructs in the ADL. This construction should offer an adjustment of the properties of a component, and should be explicitly modelled. Keeping in mind our goal to arrive at a simple model to use in a variety of circumstances, we decide to have a limited support for evolution, by offering generalisation of a component. This means that a specialisation (other way around) can exist to another component with the same shared characteristics, but also some changed properties. Multiple specialisation relations can exist per component, but only one generalisation relation can exist per component.

Support for the management of a development process

The support of management functions in the development process often relies heavily on the architecture of a system. As pointed out, the architecture is often a key artefact in dividing the workforce of a development team. The so called work breakdown structure relies heavily on the decomposition of modules to be assigned to people to implement.

Another kind of management lies in the deployment of the software on a physical hardware system. This is often described in an allocation view, which needs to be supported by a architectural model. Related to this is the assignment of modules to physical files in which the module is implemented. This is another example of supportive relations to other entities in order to support the management of the development process. This is why support for a relation between modules and artefacts outside the system will be included into our metamodel. This kind of construction is highly supported by the discussed literature. Both Kruchten [Kruchten'95] and Bass [Bass'03] use similar constructs. In the work by Bass, this is supported by the allocation structures and in the work of Kruchten by the development and physical views.

Support for dynamic interaction of architectural elements

Next to the modelling of static functional behaviour of a system, the systems runtime behaviour is also often modelled as architecture. We have seen that Kruchten [Kruchten'95], also acknowledged the need for modelling an executable unit. The runtime units can later on be mapped upon physical hardware. Also the work of Bass [Bass'03] identifies the need for runtime components, trough his 'component and connector structure' view.

In order for these kinds of models to be properly modelled, support for this will be built in the metamodel. This is why a distinction needs to be made between static and runtime components. The relations between the runtime components can vary from simple message passing to parallel composition of two runtime components.

Explicit support for decomposition

Functional units in an architecture are almost always decomposed in smaller units in order for details to be added to the component. The decomposed element(s) often give more detail of the nature than the original (higher level) component. Because this is a very common structure to use, explicit support for this should be available. Decomposition structures are found in almost all the discussed literature sources such as [Bass'03], [Kruchten'95]. Furthermore it is highly supported by the concept of separation of concern as used in [Tekinerdogan'00] to decompose a problem into sub problems. Sub problems will often be mapped onto a decomposition of requirements. These will in turn be mapped onto a decomposing of architectural elements. This is why we support this concept explicitly.

Interfaces and Constraints

Every component and connector should have at least their interface specified. This is important to the outside world, because it determines the functionality that component or connector can offer. It is also used to indicate that a component needs a services (from another component) to function correctly. Medvidovic states that without interfaces, a notation could just be a collection of boxes and lines [Medvidovic'97]. He states that only with interfaces specified, any kind of information about the architecture can be inferred. The work by [Bass'03] also identified that the main purpose of an architecture is to be used as abstraction. Internal details of elements should be hidden, and the focus should be on the interfaces that element have with other elements. We will therefore include interfaces into our metamodel. Next to an interface, the constraints that are imposed on components, connectors but also entire configurations must be recognisable in our metamodel. The main reason to include it is that the work of [Medvidovic'97] mentions them explicitly as a attribute of components, connectors and configurations.

3.3.2 The suggested architectural metamodel.

In this section, we will describe our suggested architectural metamodel and describe the elements that are included in the model. We will give a rationale for including the elements in our metamodel. The metamodel is depicted in Figure 17.

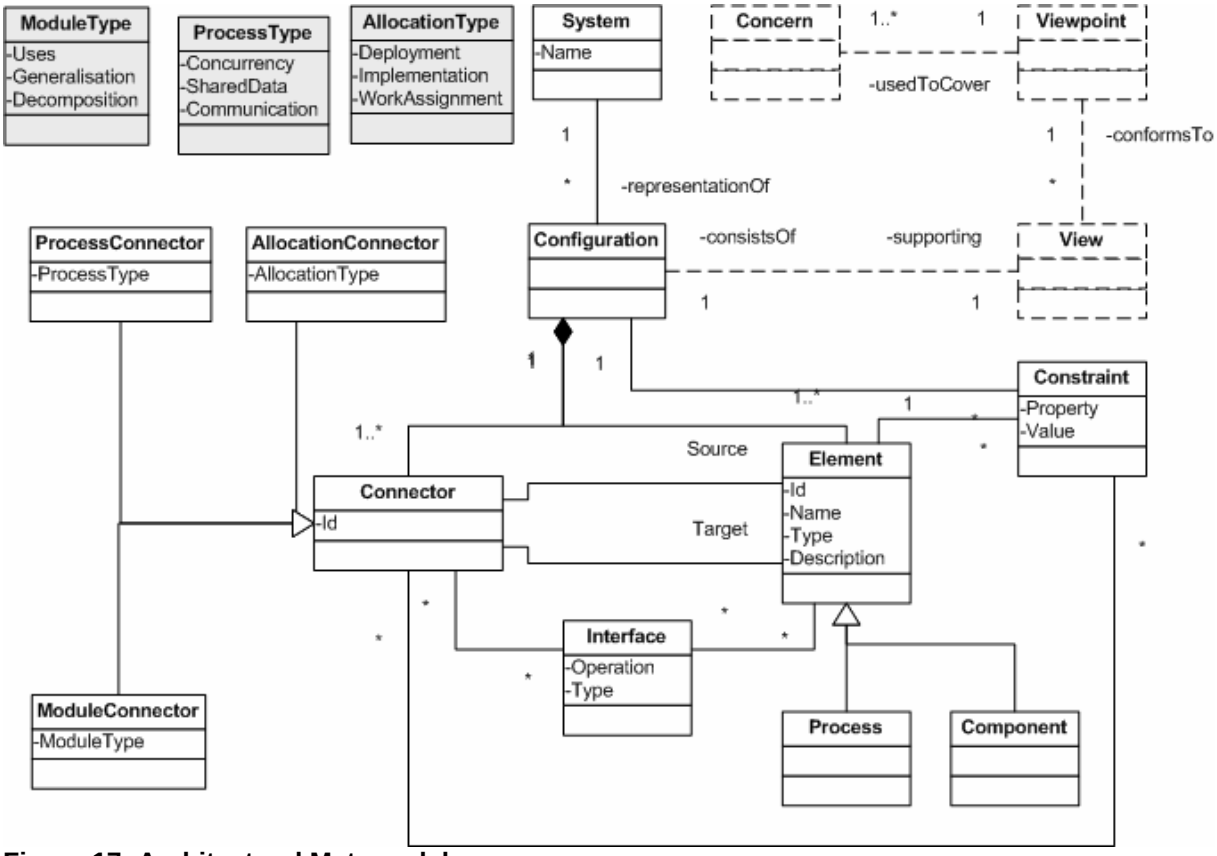


Figure 17: Architectural Metamodel

System/Configuration

The main entity in the architectural metamodel is a system. It is a conceptual entity to represent 'the system' as a whole. The system is represented by a number of configurations. These configurations describe a certain structure of the architecture. Each configuration represents a different structure of the system. A configuration is also a central element in the ADL comparison framework of [Medvidovic'97]. We have chosen to represent a system in multiple configurations on order to fulfil the definition of architecture by [Bass'03], stating that a system a system has multiple structures.

In this respect we can see high similarities between a configuration and a view, as seen in the work by [IEEE1471'00]. A view represents the system from the perspective of a particular concern. This is also true for the configurations seen in our metamodel. Views are related to viewpoints. A viewpoint describes how to structure a view, in order to satisfy a concern. Views are in that respect more or less on the same abstraction level as configurations. We do not claim that a configuration, as used in the ADL classification framework, is exactly the same as a view in the work of [IEEE1471'00]. But because there are high similarities between the two, we have depicted them in our model. It is meant as an illustrative example of how to relate configurations to the theory describing views and viewpoints. Because views and viewpoints are not a part of our metamodel we have depicted them as dashed elements.

The configuring element is moreover a kind of container that contains the different connector and components (called elements). A configuration is in this way a collection of connected components to describe the systems architecture from a certain perspective. Constraints are attached to a

configuration to enable the modeller to specify system-wide constraints, not specified in the components.

Elements

The elements of the metamodel describe the actual components that make up the architecture. Elements in our metamodel are similar to components in the ADL classification framework. In our metamodel we make a distinction between static and dynamic elements. This distinction is also made in the work of [Bass'03]. Here a distinction is made between modules and runtime components. The modules represent the implementation units, and the runtime components represent the executable behaviour of the system. We have included these elements in our metamodel following our selection criteria, stated earlier. In our metamodel, we will use the following elements:

- Components: A component is a static element representing an implementation unit of the system. A component is assigned a functional responsibility of the system.
- Process: A process is a dynamic element representing the executable behaviour of the system. Processes are explicitly treated differently because they often have different relations associated with them. These dynamic elements or processes are often found in diagrams such as a state charts.

Every element has possible constraints attached to it. These describe (Non-Functional) constraints of an element. They are a representation of the constraints as seen in the ADL classification framework. Every element also has an interface that specifies its external visible behaviour. This models the services that an elements offers, and the services it needs from other components. Both constraints and interfaces are explicitly included by our selection criteria.

Connectors

The connectors establish a relation between two elements (process or component). A connector is always connected to minimally two components. The interface of a connector is dependent on its connected components. The connector itself doesn't offer any services, but rather relays the services that the connected components have. A connector can also have constraints attached to it, like for example the multiplicity of the components that are attached. We have included constraints and interfaces to our connectors by again following our selection criteria.

The connectors can be divided into three major categories. We will shortly discuss the meaning of each of these connectors and the difference in the types that are possible.

AllocationConnector

An allocationConnector is a relation between an element of the system and elements representing artefacts outside the system. This kind of connector is specially aimed at supporting the management of the development process. An example of the use of this connector is the assignment of a component to work teams or a mapping of components to hardware components. This kind of connector supports the allocation views as defined by [Bass'03].

ProcessConnector

A processConnector is a relation between dynamic components of the system. This connector is aimed at connecting the runtime components (called processes) of a system. It has a type to indicate the meaning of the relation between the processes. The following types exist:
- Concurrency: A concurrency ProcessConnector is a relation between processes that can run in parallel with each other.
- Shared data: A shared data ProcessConnector is a relation between a process and a data source that the process uses. This can be used for performance analysis of the data source.
- Communication: A communication ProcessConnector a relation between two processes that exchange data with each other. These are often function calls to other processes.

These relations are also found in the views of [Bass'03], describing the runtime components of a system. It is included to fulfil the criteria of support for dynamic interactions of architectural elements.

ModuleConnector

A moduleConnector is a relation between static components of the system. The module connector is designed to capture the structural relations that exist among static components. These components are often seen in architectural descriptions to model the high level structure of a system. We have defined a number of types of the moduleConnector

The following types of exist:
- Decomposition: A decomposition moduleConnector is a relation between two components that have a part/whole structure. The whole is an abstraction of its parts. It is used to build hierarchies of components. This connector supports the natural problem solving process. Decomposition is performed to break down bigger components in smaller more detailed sub components.
- Uses: A uses moduleConnector is a relation between two components in which one component needs functions of the other component in order to function fully.
- Generalisation: A generalisation moduleConnector is a relation between two components where one component inherits the behaviour of another, but changes some of its properties. This kind of connector is present to enable the evolution of a component.

These relations are also found in the view of [Bass'03] describing the units of implementation of a system.

3.4 Summary

This chapter introduced the concepts of architectural modelling. We first defined the term architecture by discussing the elements of definitions found in a number of literature sources. We have identified the important role that an architecture plays in a software development project. Knowing what an architecture is and its importance in software development, we discussed how an architect arrives at an architecture that is acceptable for every stakeholder. Related to this discussion we have clarified the use of architectural views. Several independent resources describing architectural views are discussed.

Two modern system development methods, used to arrive at an architecture of a software system are discussed. Both the Rational Unified Process, and the Synthesis based Software Architecture Design method are discussed.

After discussing the concepts and processes of architectural design, we discussed the use of ADLs to capture the structure of architecture. We have done so, by discussing a research paper defining an ADL classification and comparison framework. This framework represents the elements found in 11 ADLs. The elements in this framework and the literature on architectural design are used to derive a common architectural metamodel. In order to do so, we have defined criteria for selecting the elements to put into our metamodel.

Finally, we described the resulting architectural metamodel. This metamodel is highly based upon the main categories of the ADL classification and comparison framework; Components, Connectors and Configurations.

4. Trace relations

In this chapter, we will introduce the concept of a trace relationship, or simply a trace. We are fully aware of the fact that this concept is used differently in varying circumstances and literature sources. This chapter will define what we will consider a trace, and how we will use it. First we will discuss a few of the definitions found in the literature in Section 4.1. After this we will discuss the importance of maintaining traceability and the advantages it can offer in Section 4.2. Hereafter, in Section 4.3, we will look into the concepts that play an important role in defining traceability, referring to existing literature. After these concepts have been discussed, Section 4.4 will give our own definitions of the concepts in traceability as used in this thesis. Once these concepts have been defined we will introduce the concept of a trace as we will use it throughout this thesis in Section 4.5. Section 4.6 introduces the metamodel we use to capture trace information. This Section will also give formal definitions of the relations that are used in the metamodel. Using these definitions we will discuss the concept of consistency and give definitions for consistency between models in Section 4.7. Section 4.8 will lastly discuss the difficulties in maintaining traceability. This will naturally introduce the need for a better way of managing traceability which will be covered in the next chapters.

4.1. Introduction to traceability

In order to give an introduction on the concept 'traceability', we will give an overview of a selection of definitions found in literature. One of the earlier definitions of traceability is given by [Gotel'94], in which he defined a requirement trace, distilled from several other sources, (IEEE-830 standard and the Oxford dictionary) as:

"*The ability to describe and follow the life of a requirement, in both a foreword and backward direction. (i.e., from its origins, through its development and specification, to its subsequent deployment and use, and through all periods of on going refinement and iteration in any of these phases)*"

They key issue, is that it describes the life of a requirement in both foreword and backward direction. The fact that they mention forward and backward directions hints at a direction in the development process that can be identified. This in turn hints at a special kind of relations in a development process, often called interlevel relationships, defined between development phases. Next to this, it is interesting to note that it also includes the refinement and iterations within a development phase. These relations are often denoted as intralevel relationships, being concerned with relationships amongst artefacts in the same development phase. This definition is not bound to only one type of relation. (We will give a precise definition of the types of relations that we distinguish later on in this chapter) One issue with this definition is however that it is defined for a requirement trace in particular. In order for the definition to be useful in the context of this thesis, the definition should be more general and should include tracing of other then requirement artefacts. Although it is possible to transform this definition into a general usable definition for traceability we choose not to do this. The main reason for doing this is the availability of other more recent definition of traces, becoming widely accepted by the software engineering community.

One of these definitions is defined by the organisation developing standards in the world of software engineering; the IEEE. The IEEE has standardized the definition of traceability in the IEEE 610 standard as:

"*The degree to which a relationship can be established between two or more products of the development process, especially products having a predecessor/successor or master-subordinate relationship to one another; for example, the degree to which the requirements and design of a given software component match.*"

This definition is more general than the definition by Gotel, in the fact that it does not mention requirements as the artefact to relate to. This definition is therefore more suitable for use in our context. As we can see by looking at these two definitions only, we already see that concepts like the products to which the traceability applies, and the meaning of the concept traceability itself differs. In the next sections we will investigate these concepts and also introduce a new way at looking at traceability.

4.2 Application of Traceability

Before understanding and defining what traceability is, it is a good idea to describe why we are interested in traceability in the first place. One reason is the increasing interest from the research community. Multiple reasons for the growing interest in traceability are identified in [Clevland'03]. Cleveland mentions the improved support for requirements validation, the possibility for test case generation, the recording of rationales, the possibility to perform impact analysis and protection against gold plating (over designing) as a direct result of maintaining traceability links between requirements and other artefacts. Marshall [Marshall'03] mentions the importance of tracing requirements to implementations as a mean to perform impact analysis when changing requirements or components in a development project. A final source highlighting the importance of traceability is the work by [Koning'02], in which the importance of up to date documentation is stressed. Lack of up to date documentation is identified as the cause of difficulties when adjusting an existing system. Maintaining a close link between implementation and documentation (such as requirement and architectural models) is an important factor when modifying existing systems.

The purpose of maintaining traces between artefacts in a development process is very dependent on the interest or goal of the stakeholder [Knethen'02]. This is also one of the problems when discussing traceability; the meaning of traceability differs with the goal that people have in mind with maintaining the traces. We will give some examples of the usage of traceability:

- A customer of a software product will be interested in the satisfaction of the requirements in the software product being built.
- A software designer will be interested in checking whether the requirements are implemented in design components and that no unneeded design is present.
- A software tester will be interested in checking whether all the requirements have an associated test case.

All the interests of these stakeholders can be met by maintaining traceability. This is however highly dependent on the information that is captured in these traces. The information to be captured in the traces is as a result highly dependent on the goals of the stakeholders. The goal of the stakeholder in the traceability process will as largely determine the trace model that will be used (what information will be included in the traces).

Apart from the importance identified above, we ourselves are of course also interested in investigating traceability; Traceability is the basic mechanism frequently used to keep requirements and other design artefacts in sync; Being the main subject of this thesis. This is backed up by the work of [Cleveland'03], who states that traceability techniques are currently mostly used (foremost by software configuration management tools) to identify all artefacts that should be updated when a change to a requirement is introduced (when the synchronisation of development artefacts is most at risk). It is therefore important to first define what traceability is, and how to establish it in the context of requirements and architectural design.

4.3 What is traceability

What do we consider as a trace relation? The first thing notice, by looking at the term 'trace relationship' is the observation that we are talking about a relationship. This means that it, in some way, relates entities to each other. In previous chapters, we have already seen multiple relationships such as relations amongst requirements and architectural components. These all relate multiple elements together. We however do not automatically consider these as traces. But then, what is a trace? The question 'what is a trace' is highly dependent on a number of factors. We have already mentioned the goals of the stakeholders. We will return to this later, when we compose our working definition of a trace. Next to this, it is also highly dependent on the entities, and the relationships that we want to trace. Following the work of [Knethen'02], identifying the "core concepts" in tracing approaches, we will discuss the concepts of traceability in the following sections. We will in these sections use the terminology used by Knethen.

4.3.1 Elements in a trace

Closely related to the goal of a trace are the elements that are connected by the trace. The selection of the elements to include in a trace will often mainly be driven by the reason of its creation (the goal of the stakeholder). A trace relation, created to check testability, will include other elements than a trace relation created to check the implementation of requirements. This dependency between the goal of a trace link and the elements to use in this relation is recognised in [Knethen'02].

Granularity

Another aspect of an element in a trace relation is the granularity of that element. In both the earlier discussed metamodels (requirement and architectural metamodel) we have seen that decomposition is used as a mechanism to influence the granularity. In this way a coarse grained element can be detailed into a fine grained level. Trace relations can also exist on these different levels of granularity. This (again) depends on the intended stakeholder of the trace relation. Sometimes a programmer might want to trace a particular function to the requirement from which it originated, but on the other hand, a high level system designer might just want to trace the coarsest level of architecture to the second level of refinement of that architecture. Because there is a difference in the granularity of the elements, this also indicates differences in the details of the trace. The detail of the information that is encapsulated in the trace (and it's elements) is an important aspect of traceability. Approaches for supporting traceability on different levels of granularity currently exist. An example of a fine grained level of traceability is described in the work of [Lange'97]. An interaction graph of objects in a Object Oriented program are traced, in order to improve the understanding of a new developer. Here the tracing takes place on the level of source code. On the other end of the granularity scale we find techniques used by e.g. RequisitePro tracing requirement documents to design documents. Choosing the right level of granularity is important in order to best suit the goals of the traceability stakeholders. This is a challenging area of work, demonstrated by the work of Bohner and Arnold, mentioned in [Knethen'02]. They state that current traceability tools often offer a too coarse level of granularity to support accurate impact analysis.

Attributes

Following the work by Knethen, we also have to mention the characteristics of the traced elements. In the work by Knethen these are mentioned as attributes of the elements. Attributes are important in a trace relation because they dictate what information can be captured in the trace. If the name attribute of a stakeholder element is missing, the identification of the correct stakeholder after a 'contradicting stakeholder trace' would become very difficult. The work by Knethen gives an overview of the most important attributes that are used in industrial projects. We will summarize these attributes:

- Indication of the difficulty in implementing a requirement
- Effort needed to implement a function
- Financers of the requirement
- Satisfaction information to support testing
- Priority
- Release in which the implementation is planned
- Source document
- Status of the requirement
- Decomposition level
- Type according to a certain taxonomy.

We have chosen to implement most of these attributes in our models described in previous chapter. Because our aim is simplicity in models we have not included all of the abovementioned attributes. We mention them however for completeness.

Elements in our study

For completeness we will repeat what elements we will include in our trace relations. This is an important aspect of the trace relations because it will dictate which traces we can be implemented:

- Requirement Elements:
  - Requirements, External objects, Rationale, Stakeholder
- Architectural Elements:
  - Configurations, Elements (Processes & Components), Interfaces, Constraints, External Objects.

4.3.2 Relations in a trace

The other important elements in a tracing are the relations that can be maintained between the elements in a software development project. Currently there is little consensus on the kind of trace relations that should be captured in a software development project [Knethen'02]. In this section we will discuss currently used trace relations, by following the way of working of Knethen. We will discuss subsequently: the kinds of trace relationships that exist, the direction of the trace relation, the attributes of a trace to capture, the creation of relationships and the representation of the relations. When discussing them, we will use the terminology of Knethen. We will later on define our own definitions.

Kind of relationships

The work by Knethen distinguished two types of relations; relations on the same abstraction, and relations on different abstractions. Abstractions roughly translate to abstractions levels, or levels of detail. Next to these kinds of relations, another kind of relation can be distinguished, based on the versions of the artefacts in a document. These are often called evolutionary links or relationships. We will discuss all these kinds of relations and their variants and uses in practice below.

Relations on the same abstraction.

Representation relationships

These relations are used to indicate that different models represent different views on the same information. We have seen the term view earlier in discussing the architectural models. Often different views on the same architecture are made to support different stakeholders. These views represent the same system, and are as a result related. Representation relations are used to connect these different views of the system. An example of elements that can be related are textual requirement documents and use case diagrams, both describing the same required behaviour of the system. A mapping between these different representations can be made.

A key difficulty in creating these kinds of relations is the maintenance of consistency between the different models representing the same information. Different solutions to this problem are summarized in [Knethen'02]. They mention the work of Finkelstein, in which classical logic is used as a common language to map to. Another solution is described in the work of Pohl, which uses a three dimensional framework for requirement management. Here the Information is captured along a different axle as the representation, which is separated in this way. A last well known method to cope with inconsistencies amongst views is the use of a metamodel to describe the relationships between different diagram types. This approach is taken by the Four layer model architecture of UML which consists of user objects, models, metamodels and meta metamodels .

Dependency relationships

These relations relate models on the same abstraction levels, but do not represent the same information. These relations relate entities that depend on each other, but represent other logical entities. A good example of such a relation is a (implicit) relation that can exist between source code blocks created by for example function calls. This relates two blocks, because they exchange data or call upon each other. These two blocks are clearly on the same abstraction level. These relations can often be detected automatically by analysing the code of a program.

On a higher design level we also come across dependency relations. These relations are found in the interaction between high level components, describing the high level behaviour of the system. These components are related each but represent different logical entities. The relationships between them fall into the category of dependency relationships. The modelling of these dependency relations often implies the specification of interfaces between these components, opening the way for component based development.

Relationships between abstractions

These relations refer to the relationships created between different abstractions. Within this category of relationships two types of these relations are distinguished. We will discuss these two kinds hereafter.

Within-level refinement relationships

This kind of relation refers to a relation between entities situated on different abstractions (detail level), but within one (development) level. An example of such a relation is the relation created between two test cases, in which on test case contains a more detailed test of a specific situation than the other element. Both are test cases, but the detail level differs. Specifically, [Knethen'02] mentions three situations in which these relations are found. Firstly, in goal-based requirements engineering goals are decomposed in a hierarchy of goals until requirements are found to support these goals. The relations between the goals and requirements can be considered within-level refinement relationships. Secondly, refinement, found in a development approaches can also be regarded as a relation in this category. Lastly decomposition can also be regarded as being a relation in this category. These decomposition relations are used to be able to traverse a requirement hierarchy.

Between-level refinement relationships

These relations refer to those relations created between entities at different abstractions and between different abstraction (development) levels. This is the kind of relations often seen as interlevel relationships. An example of such a relation could be the relation between a requirement and a high level design component. The work by [Knethen'02] discusses several approaches that use between-level refinement relationships. First again the work of Pohl's three dimensional requirements engineering framework is mentioned. Specifically, the specification axle of this framework covers these relations.

Next to this, also the four variable model, by Parnas et al, can be considered to be using between-level refinement relationships. In this model, the system is described by relations between four sets of quantities; monitored environmental, controlled environmental, input and output quantity. This model is foremost aimed at describing controlled systems. The monitored environmental quantities represent the influences on the system, the controlled environmental quantities represent the manipulation by the system. The input quantities are the values given by the sensors of the system and the output quantities are the output values given to the actuators. The behaviour is, described by relations between these quantities. The requirements of a system are for example described by combining two relations: NAT and REQ, both being relation between the monitored and controlled environmental quantities. NAT describes the (natural) boundaries of the system and REQ described the behaviour of the system. The system design (another abstraction level also) is described by the relation between IN and OUT. IN is a relation between monitored and input quantities. OUT is a relation between output and controlled quantities. The last relation that is described in this model is the SOF relation, which describes the software requirements by describing the relation between input and output quantities.

Another example of the presence of this relation, given by [Knethen'02], is the gap that can exist between development phases in a development approach. It is stated that when a system could be completely generated from a system specification, that there would be no gap between the phases and the problems with between-level refinement would be largely reduced. As stated, this is currently however not very much the case. It is claimed that object oriented approaches would reduce this gap, removing the need for this kind of relations. Largely because object oriented design should be closer to the object oriented development used nowadays.

Lastly, requirement traceability tools are mentioned. Some of these tools offer between-level refinement by for example creating relations between requirement documents and design elements.

In [Knethen'02], a summary of currently used between-level refinement in practice is given. We will present a small summary of this research:

- Applicability / dependency links: These links indicate that one element is dependent on another element to function/exist. Removing an element with such a link associated implies great caution because it might imply that the related element stops functioning.
- Links between (textual) requirements and object oriented models: A number of these links exists. One kind of link that is found in practice is a link between requirements and models like source documents use cases, analysis products, tests cases etc. These links are also called rationale and evolution links because they link a requirement to a source of other analysis product. They for example ensure that every requirement is covered by a test case. Next to these links we can also distinguish links among object oriented models and to software elements. These are often used to trace a requirement to other work products like design elements or source code.
- Links between error listings and change requests in order to trace a error back to a introduced change.
- Evolution links between requirements and test specification and a link between test specification and test cases.
- Evolution links between: requirements and functional high level units, requirements and subsystems, requirements of a subsystem and validation tests.

Evolutionary relationships

We can also distinguish yet another type of link, connecting different versions of models together. These kinds of links are often called evolutionary or historical links. These links are used to trace the different versions of an element or models that have existed in a development process in order to support recovery of earlier versions. Different approaches to maintain these links exist and are discussed in the work of [Knethen'02]. First the three dimensional framework for requirements engineering of Pohl uses the agreement axle to cover the different versions of documents. Another approach, called REMAP, by Ramesh and Dhar uses a prototype environment that provide assistance to stakeholders when designing and managing large systems, supporting evolution in this design. Finally, ESE is another system which is basically a research environment to support evolution of software documents is mentioned. In ESE, each document can have multiple versions and links to other documents. In practice the most used evolutionary links use inheritance to indicate a change in a element like new, changed or deleted elements. Because these kind of relations involve a rather new field of study, and would require extensive addition study, we choose not to include this relation in our study.

The direction of a relation

As mentioned in the beginning of this chapter, one well known definition of requirements traceability is the one by Gotel and Finkelstein. It divides the requirement traceability process in two parts: pre requirements specification (pre-RS) traceability and post-requirement specification (post-RS) traceability. The pre-RS traceability refers to the traceability of a requirement to documents/elements explaining the requirement or the reason behind it (rationale). Post-RS traceability refers to the tracing of a requirement to the lower level design documents which leads to implementation.

As we already identified earlier in this chapter, Gotel and Finkelstein also make a distinction in the forward and backward direction that can be discovered when following a requirement through its development phases. The distinction between forward and backward traceability does not take the perspective of a single entity (before and after this entity), but rather indicates the traceability direction through the development (abstraction) levels. As seen in Figure 18 forward traceability is aimed at traceability from domain problems and requirements towards implementation entities. Backwards traceability refers to the maintenance of links from an implementation element to its originating design and requirements.

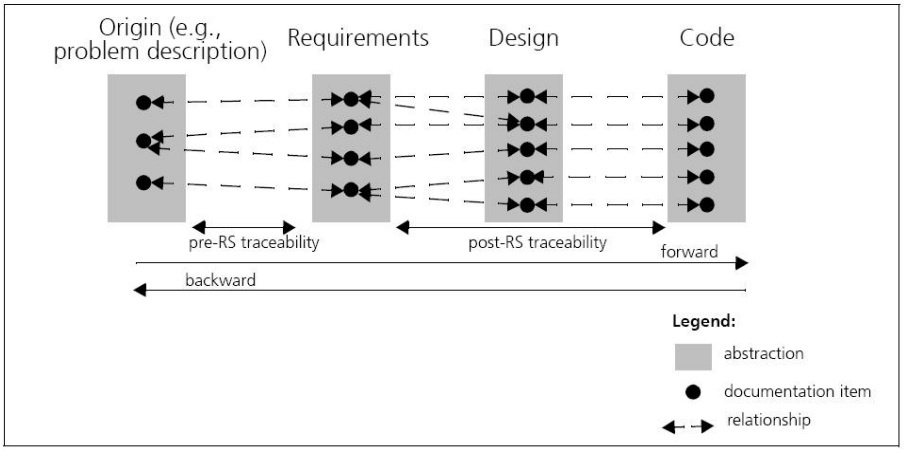


Figure 18: Forward and backward traceability[Knethen'02]

Often it is argued that traceability should not be aimed at only one direction, but should be in both directions, called bidirectional. Currently most of the traceability methods/tools support some sort of post-requirement traceability. Pre-requirement traceability is however often neglected or not supported. Two methods that do support this are a method by Gotel and Finkelstein themselves [Gotel'95], in which they try to involve people in the Requirement Traceability process. Another approach which mentions Pre-RS traceability is the Requirement Engineering Framework of Pohl.

Attributes of a relationship.

It is possible to include additional attributes in a relation. In literature there is little mentioning of which attributes to include. One attribute that we have seen earlier in the requirement metamodel is the rationale. This is an important attribute to include because it can record the reasons behind certain decisions. This can be important in the maintenance of the system when a engineer needs to understand a system. A rationale is more focussed on process related elements in a development trajectory. Traceability is more focussed at the structure of a product and to be able to understand the implications of change.

Next to a design rationale, a number of other attributes of a relationship can be identified when looking at the current practice. We give a small summary of the most well known:

- Active: indicating whether a link is currently used or not
- Status: indicating whether a implementation is finished or not.
- Description: When a relation is not completely clear or needs extra explanation this can be recorded here.
- Completion date: indicates when a relation can be considered fully operational. I.e. when the component is implemented to fulfil a requirement, this can be indicated in this attribute.

The creation of relationships

A (trace) relationship does not just exist. It has to be specified. [Knethen'02] Identifies two ways of creating a relationship: either implicit or explicit.

Implicit relationships are more or less created automatically by implying some rules on how and when to create them. They do not require a person to manually create them. We will discuss the mechanism on how to identify/ and create relationships in an implicit way:

- Name tracing: Traces can be established when names and abbreviations are used in similar ways. The problem with name tracing is that it is highly dependent on naming conventions, especially in a large development project. This kind of tracing is seen in tools that support representation relations (e.g. requirement specification and a use case with the same name, describing the same system
- Relationships given by structure: Another way of creating an implicit trace relation is by the structure of the model. An example of creating such a trace is the creation of a trace by following the decomposition. A top-level requirement can be followed to the smallest child in that model,

creating trace relations between every parent/child pair. These kind of relations are often only created in case of within-level relationships.

- Relationships given by modelling paradigm: Some modelling paradigm offer automatically provided relations between modelling elements. [Knethen'02] mentions an example in which relations between use cases and sequence diagrams are automatically created. How this is achieved is not explained however.
- Dynamic relationships between code components: Implicit relation can be created by carrying out dependency analysis. An example of this kind of creation is the analysis of code to create trace links between code blocks.

On the other end of the spectrum we find explicit created relationships, which are manually created between artefacts. It requires an architect or developer to select the elements and explicitly create a relation between them. A developer can for example create a relation between a element of the design and a code block that implements it. These relations are created by external decisions of a stakeholder. It will depend on the stakeholder, which relations are created. To support the creation of these relations requirement management tools are available. These tools ease the stakeholder to (graphically) create these relations. Some modelling tools also offer the explicit creation of relationship in for example UML models. These however only aid the person responsible for the relations. It remains up to the person responsible for them to actually create them.

Representation of relations.

Whether trace relations are created implicitly or explicitly, in order for them to be meaningful they need to be represented in a convenient manner. Wieringa [Wieringa'05] summarizes a number of graphical ways to represent these trace relations:

- Traceability matrix: A matrix can be used to express the links between modelling entities. The entities are projected on the horizontal and vertical axles. Entries at the cross section of these axes represent the presence of trace relations.
- Graphical models: Another way of representing traces is the explicit modelling of elements and relationships as lines between them. This is the same approach used to model the relations in both the requirements and architectural models
- Cross-reference: Relationships can also be created by using reference to other elements. In text documents these can be references often found in research papers. Cross references help to make the document navigated through.

Most current requirement management tools offer these representations in order to display the trace relations in place.

4.4 Categorisation of relations

Having discussed the types of relations seen in current literature, we will now describe the terms we will use throughout the remainder of the thesis for distinguishing between types of relations.

A common way to categorize the kinds of relations is according to the levels of abstractions which they connect. Interlevel relations often refer to relations within the same level of abstraction. Intralevel relations often refer to relations between different levels of abstraction. We have already shortly mentioned the distinction that is made between inter and intra as a prefix before. We also use these prefixes when referring to relations. The problem lies in the term 'abstractions'. Abstractions are used very differently among different literature sources. Because of this lack of consensus in literature, we give our own definitions of interlevel and intralevel relations.

- An interlevel relation is a relation between model elements and/or models in different development phases within the same iteration of a software development process.

An example of such a relation is a relation between a requirement model, and a design model. These Models are created in different development phases, and are created in the same iteration of development. Because they are created within one iteration, these models will often contain the same level of detail.

- An intralevel relation is a relation between model elements and/or models in different iterations within the same development phase of a software development process.

An example of such a relation is a relation between two requirement models, each created in a different iteration of development. These models are created in different iterations of the development process, but are both created in the same development phase (requirements phase). Because they are created between different iterations of a development process, they will often connect elements on a different level of detail.

We graphically depict this in Figure 19. Here we assume that the Iterations levels are depicted vertically and development phases horizontally. From now on, we will not use the terms horizontal or vertical, because it can easily create confusion by rotating a diagram by 90 degrees. (Depicting the iterations horizontally). We will instead use the terms inter and intra level relations. In Figure 19, it is clearly visible that relations between development phases are called interlevel relations. Intralevel relations remain within one development phase, but relate different Iterations of development. The term Iterations that we use is related to iterations we saw earlier when discussing RUP. RUP however also uses the term phases, relating to segmentation over time. These are clearly different from what we call development phases. What we call development phases refers to the process disciplines seen in RUP.

Inter and Intralevel relations are also commonly called between-model relations, because they relate two different models. We will not use this terminology. However, to distinguish between relations between models (intra and inter level), and the relations found in our metamodels (within a model), we will call the later, within-model relations. These refer to the relations as found in for example the requirement metamodel.

- A within-model relation is a relation between elements in a model in one single iteration and in one single development phase of a software development process.

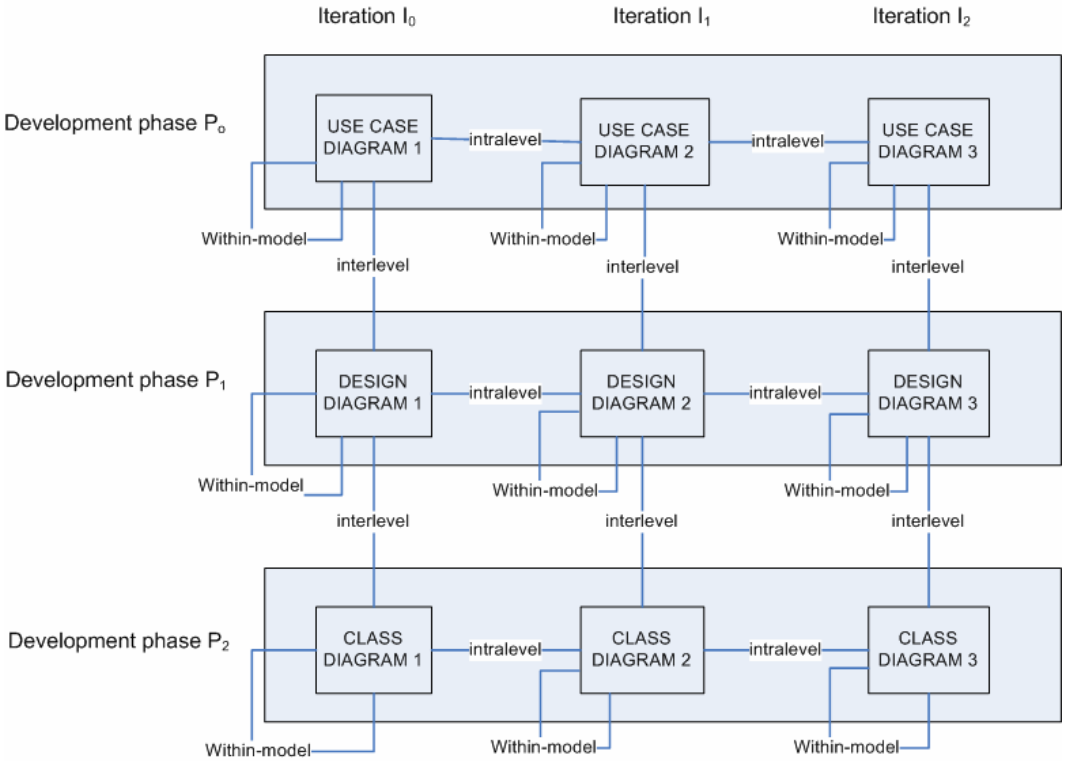


Figure 19: Types of relations

4.5 Traceability as a role

The within-model relations that we have seen in previous chapters have all one common characteristic; they are all relations between elements contained in one development phase. The relations defined between the requirements in the requirement metamodel for example, relate requirements to other requirements or artefacts in the requirement phase. The same thing can be said of the architectural metamodel. Commonly these within-model relations are not seen as trace relations. This is often reserved for relating high level development artefact through the development phases towards realisation. When we look at the definition given in the previous section, by the IEEE, this does however not always need to be the case.

An example of the use of within-model relationships in tracing is the use of decomposition. In this trace, a requirement is followed in its natural problem solving breakdown structure until it is small enough to be linked to a design level artefact. A lot of industry companies actually work like this, by starting with one top-level requirement description, and decomposing (detailing) it until sub requirements are understandable enough for programmers / designers to implement. Here we clearly see an example of a situation in which an within-model relationship (decomposition) is a (part of) a trace relation.

This does however not mean that every relation that has been discussed until now is also per definition a trace relation. When we look for example at the cost influence relation that can exist between two requirements, this will often not be considered as a trace link on its own. It is merely a relation that indicates that two requirements influence the implementation costs of each other. It becomes however another story when the cost influence relation is included in a wider analysis of costs per requirements (with the accountant as a stakeholder). Then people would tend to see it as traceability of costs.

What we really see from the abovementioned is that the content of a trace relation is highly dependent on the goal that people have. I.e. what do people want to trace. A trace is as a result often formed to serve the people that are interested in specific information. A result of this observation is that it is very hard to precisely define what a trace exactly is; it depends on the specific situation at hand.

Instead of forming a long list of what we consider a trace, we decide to treat a trace as a role of a relation/relations. A relationship is not always a trace relation, but can be treated as one if needed. This enables us to treat interlevel, intralevel and within-model relations as trace relations.

Our goal in this thesis is to allow consistency checking between requirements and architectural artefacts. In order to perform the consistency checking we will need to establish interlevel relationships between the requirement level and architectural design. We will capture these relations in a separate trace metamodel, discussed in the next section.

4.6 Towards a traceability metamodel.

In the previous section we have investigated what we will consider a trace relation in the setting of this thesis. We will as a result from now on treat a trace relation as a role of existing relations (as for instance defined in the previous chapters about requirement and architectural modelling). We can now treat the within-model relationships already defined in the metamodels of previous chapters as trace links.

We lack however the ability to also define a relationship between development phases and between iterations. These relations, called interlevel and intralevel relationships respectively, need to be defined in order to treat them as trace relations. The importance of these relations is indicated by the facts that many literature sources often mention these relations specifically as being trace links. They are for example used to track an artefact from the early development phases to the implementation in a later development phase.

In order to be able to establish these relations and also treat them as traceability relations, we have to have a metamodel for defining trace relations. Because this thesis focuses on the consistency checking between requirement and design artefacts (modelled according to the architectural metamodel), we will define a trace metamodel focussed at establishing relationships between development phases. Below we fist present the trace metamodel [Figure 20] and discuss it subsequently.

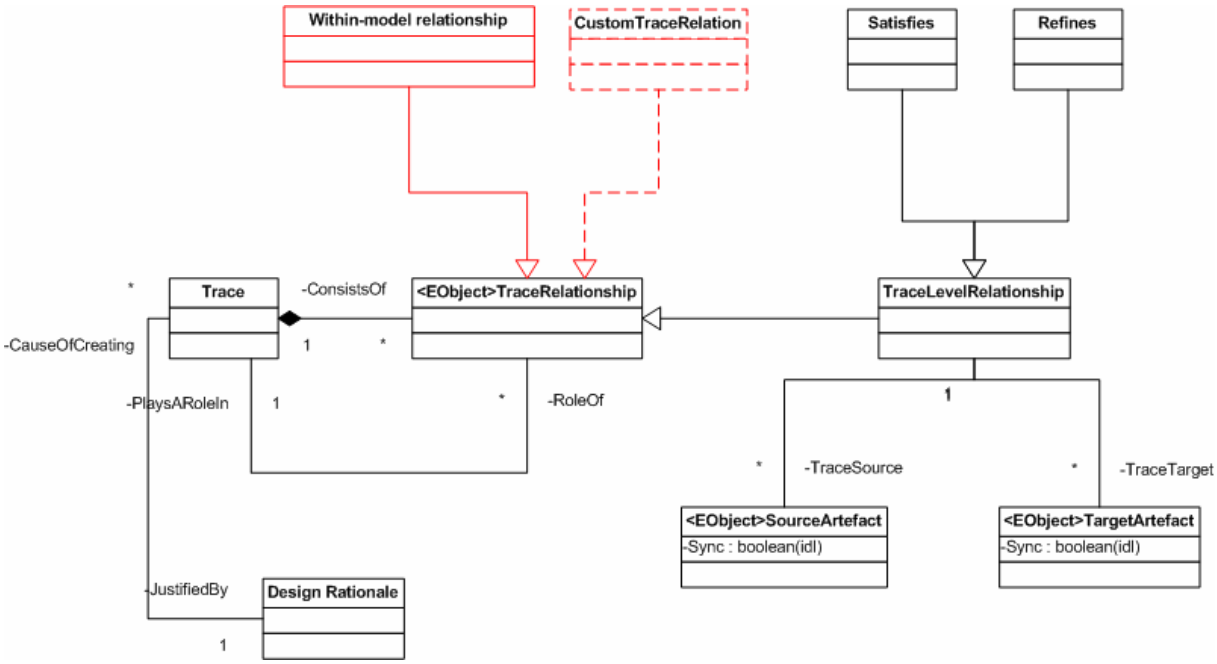


Figure 20: Trace metamodel

The central entity in this diagram is the trace element which represents the actual trace link between the related entities. Because a trace link is always created to serve a specific stakeholder, a design rationale is connected to it in order to allow for justification of this link, e.g. why it is created. A trace consists of a number of relations which are called trace relationships. These trace relationships play an important role in the actual tracing. We have introduced this trace relationship entity to allow for derivation of specific relationship entities in multiple other models (like the relations in the requirement and architectural metamodel). This construction, as a result, allows for the (within-model) relations in other models to be included into the trace relations. The dashed custom trace relation indicates that this model can be extended using other relationships currently not included in the model.

Next to these existing relations, we also want to include other relations to be formed. For this purpose we have included a number of relations that derive from what we call TraceLevelRelations. We

acknowledge that this name is rather ambiguous. It however represents both interlevel and intralevel relations and is therefore hard capture in a single term. From the abovementioned we have derived two kinds of relations that are used in practice and currently not yet in the earlier discussed metamodels. We will discuss these relations in a point by point manner:

- Satisfies: A satisfies relation is an interlevel relation between elements in two separate development phases. One element is created to satisfy the element on a higher level. We will use this relation to create a relation between a requirement and architectural design element which satisfies this requirement. This kind of relation falls in the category of between-level relationships as defined by [Knethen'02].

- Refines: In Chapter 2, covering requirements modelling, we have already discussed the refinement relation in the setting of requirement modelling. Here, we have seen that this relation can mean different thing depending on the literature that you read. Basically we have covered a number of refinement relations that can be included in the requirement metamodel, being; the adding of details to a requirement, refining a business requirement into a software requirement and replacing a requirement with another requirement.
  We also identified that refinement can be used to refine a requirement model into other design artefacts in that same phase. When focussing on the requirement management, we stated that this kind of use of the refine relation is outside the scope and will be treated in the trace metamodel. We therefore add this kind of relation to the metamodel depicted above. With refinement we define, an intralevel relation between two models or model elements in different iterations. Hereby we establish a relation between models or model elements in the same development phase, but in different iterations of the development process. An example of the use of this relation is a situation in which a requirement created in the first iteration of development, is detailed by another requirement the next iteration. Another example can be a high level design, which is a refinement into another design with more sophisticated components. This can be the result of learning more details about the system. This kind of relation would fall in the category of within level relationships as defined by [Knethen'02].

To formalize these relations, we have to first establish a number of definitions on models and relations that can exist. We first formalize what we will call a model in definition 1.

Definition 1:	A model M is a set of elements e such that: $M = \{e^1, .., e^n\}$

A model is here just treated as a collection of model elements, comprising the model. We can now formalize what we have called inter and intra level relation throughout the thesis in definitions 2 and definition 3.

Definition 2:	A Interlevel relation is a relation r_i such that $r_i \subset M_1 \times M_2$ in which M_1 and M_2 are in the same iteration of the development process. They are however contained in separate development phases.
Definition 3:	A Intralevel relation is a relation r_e such that $r_e \subset M_1 \times M_2$, in which M_1 and M_2 are models in separate iterations of development. They are contained in one development phases.

The relations between models in different development phase will often be in the same Iteration of development (of a iterative development). Having established the difference between intra and inter level relations, we can now also establish what we call a general relation in definition 4. Based on this relation we can also formalize the concept of a trace as a role of another relation. This is formalized in definitions 4, 5 and 6.

Definition 4:	A General relation is a relation r such that $r \subset M_1 \times M_2$ in which M_1 and M_2 are models in arbitrary development phases. Furthermore: $r \subseteq r_i \cup r_e$

Definition 5: A trace relation is a relation t_r such that $t_r \subseteq r$, in which t_r is a role of r. A role of a relation allows the relation to be treated differently by assigning a special status to the relation. When a general relation is assigned as a trace relation, it will be treated as a trace relation, enabling a relation to be used in an actual Trace.

Definition 6: A trace, T, is a collection of trace relations: $T = t_r^1 \cup .. \cup t_r^2$

Example Trace: $T = \{(e^1, e^2), (e^2, e^3), ..., (e^{n-1}, e^n)\}$

A trace is formalized as the union of all relations, treated as trace relations. These relations can be either inter or intra level relations.

We can now formally define the aforementioned satisfies relation in definition 9.

Definition 8: A requirement model M_r is set of requirement elements r such that $M_r = \{r^1, .., r^n\}$.

Definition 9: A *satisfies* relation is defined as: $t_{sat} \subseteq M_r \times 2^{M_2}$, where M_2 is an arbitrary, non requirements model.

Example Satisfies Trace: $T_{sat} = \{(r^1, \{e^1, e^2, e^3\}), ..., (r^n, e^n)\}$

The following holds for the satisfies relation:

1. Every requirement must be satisfied.

For each requirement $r \in M_r$ there exists exactly one $m \in 2^M$ such that: $(r, m) \in T_{sat}^{rm}$

2. Every model element satisfies at least one requirement

For each model element $n \in M$, there exists at least one $m \in 2^M$ and $r \in M_r$ such that $n \in m$ and $(r, m) \in T_{sat}^{rm}$

The refine relation is defined in a similar way in definition 10.

Definition 10: A *refines* relation is defined as: $t_{ref} \subseteq M1 \times 2^{M_2}$, where M_1 and M_2 are arbitrary models in the same development phase, e.g. architectural models.

Example Refines Trace:
$T_{ref} = \{(e^1, \{e^3, e^4\}), (e^2, \{e^5, e^6\}), ..., (e^n, \{e^m\})\}$

Let M_1 and M_2 be models related by refinement. The following holds for the refinement relation:

1. Every model element in the source model must be refined.

For each model element $m \in M_1$, there exists at least one $M \in 2^{M_2}$ such that $(m, M) \in T_{ref}^{M_1 M_2}$

2. Every model element is derived via refinement from another one

For each model element $m_2 \in M_2$, there exists at least one $M \in 2^{M_2}$ and $m_1 \in M_1$ such that $m_2 \in M$ and $(m_1, M) \in T_{ref}^{M_1 M_2}$

3. Connected model elements in the source model lead to connected refinement in the target model.

If $m, n \in M_1$ and $connected(m, n)$ then $connected(M, N)$, where $M \in 2^{M_2}$, $N \in 2^{M_2}$ and $(m, M) \in t_{ref}^{M_1 M_2}$ and $(n, N) \in t_{ref}^{M_1 M_2}$

How to treat connected depends on the model at hand. An association between classes can indicate connectedness. Also method invocation in source code can indicate connectedness.

The Satisfies and Refines relations can be combined together to form complex relations. An example of such a construction is found in Figure 21

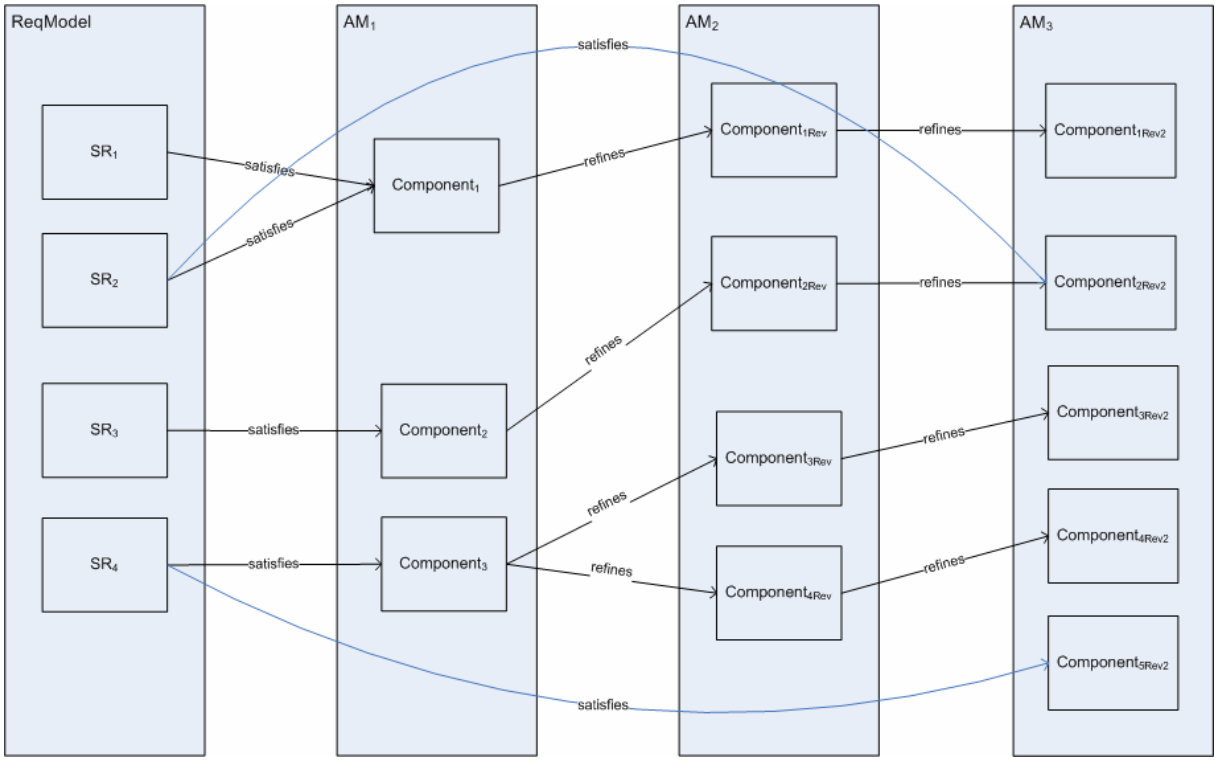


Figure 21: Satisfies and Refines Combined

What we see here is that a requirements model containing a number of requirements (SR_1 – SR_4) is initially satisfied by an architectural model AM_1. After a refinement of model AM_1 into model AM_2, the requirements will also be satisfied by the refined model AM_2. SR_4 for example is now also satisfied by both $Component_{3rev}$ and $Component_{4rev}$ in AM_2. A satisfies relation can also span multiple refinements, as seen in SR_2 and SR_4. SR_4 for example is satisfied by $Component_{3rev2}$, $Components_{4rev2}$ and $Component_{5rev2}$ in model AM_3 .These complex relations are typical for a real life system. What we see is that in order to check what elements satisfy a requirement, it is important to specify in what model that satisfying components resides.

4.7 The concept of consistency

Having now defined the satisfies and refines relations, we will use these definitions to establish how we will define consistency between models. In order to do this, we will first give a small overview of the concept of consistency in existing literature. An interesting source in this respect is the work by Hoss [Hoss'06], which offers a solution to inconsistencies that can occur between different design views. This is achieved by defining a consistency checking process, including a consistency framework, covering a number of consistency types that can exist. Although this type of consistency checking (between views) does not directly correspond to our interpretation of consistency, it offers some interesting discussions (especially the consistency framework) on existing definitions of consistency.

The first observation in the work by [Hoss'06] is that there is little consensus on the meaning of consistency amongst researchers. As a result, a number of definitions are found in the work by [Hoss'06], which each correspond to a different type of consistency in software requirements and design. This leads to the observation that the consistency conditions depend on an number of factors; The diagrams involved, the development process employed, and the current stage of the development" [Hoss'06]. Even though the term consistency will differ with these factors, it is critical to specify the term 'consistency' in a precise and formal method and there should exist an automated mechanism for verifying consistency [Hoss'06]. This observation concurs with our need for tool support for consistency checking between requirements and lower level design artefacts. Also a formal description definition of the term consistency is an important factor in this thesis, in order to automate the consistency checking in any tooling.

The work by [Hoss'06] makes a distinction between intra and inter consistency. Intra consistency is defined as consistency between diagrams (within a specific model), typically at different abstraction levels. (e.g. between two different UML diagrams, containing additional features) [Hoss'06]. Inter consistency refers to consistency between models (within a specific system) at the same level of abstraction (e.g. between a UML class diagram and a UML sequence diagram, with the same level of detail). [Hoss'06]. The use of the terms intra and inter concurs with our use of these terms in describing different relations. The keyword here is however the term 'abstraction'. We have chosen to avoid this ambiguous term.

As the title of this thesis indicates, we will only deal with one specific kind of consistency, named in the title as; Interlevel consistency between requirements and design artefacts. [Hoss'06] states these relations as relations at the same level of abstraction. This could be interpreted as the being within the same iteration. These relations are defined between elements at different phases of development (requirements modelling, architectural design, detailed design etc). Intra level consistency is defined as consistency in relations between elements at different iterations, within the same development phase, (often adding details within that phase). Hoss by [Hoss'06] here again uses the ambiguous term abstraction, which we do not.

We will as a result treat consistency checking between artefacts in different development phases. Following a definition used in a goal oriented approach, KAOS (Knowledge Acquisition in autOmated Specification) for (in) consistency as; "a set of descriptions is inconsistent if there is no way to satisfy those descriptions all together" [Hoss'06]. We will define a model to be consistent when every element in a model M1 is satisfied by (a set of) element(s) in a model M2. Because of the interlevel designation, the target and source model will be at different development phases. The definition of consistency with respect to another model regarding the satisfies relation can be found in definition 11.

	Let M_1 is a requirements model and M_2 is a non requirements-model.
Definition 11:	A model M_1 is considered consistent with respect to another model M_2, $consitent(M_1, M_2)$, regarding the satisfies relation when a satisfies relation T_{sat} can be established between M_1 and M_2.

This implies that two models, a requirement model, and a non requirement-model are only consistent when every requirement in the requirement model is satisfied, and visa versa when every element in the non requirement model has at least one originating requirement connected to it.

Consistency can also be defined regarding the refines relation, specified earlier. The definition of consistency with respect to another model regarding the refines relation can be found in definition 12.

Definition 12:	A model M_1 is considered consistent with respect to another model M_2, $consitent(M_1, M_2)$, regarding the refines relation when a refines relation T_{ref} can be established between M_1 and M_2.

This implies that two models M1 and M2 are consistent regarding refinement, if every model element in model M1 is related to a (set of) element(s) in model M2. Also every element in M2 belongs to at least one set of elements that refines an element in M1.

Now that we have defined consistency regarding both the satisfies and refines relation, we can also define how to combine these definitions into one definition stating consistency using both these relations. We will define this as follows in definition 13. It should be noted that definition 13 takes into account a series of refinement steps of architectural models thus using the transitivity property of refines.

	Assume that we have a requirement model, R and a series of architectural models $M_1, M_2, \ldots, M_n$. Every architectural model refines the preceding one.
Definition 13:	A model R and M_n are considered consistent regarding the satisfies relation if R and M_{n-1} are consistent regarding the satisfies relation and M_{n-1} and M_n are consistent regarding the refines relation.

In this case a requirement model is satisfied by an architectural model which is refined into (a number of) other architectural model(s). In this way a requirement model can be satisfied by a refinement of an architectural model that satisfies that requirement. This definition now also allows us to check the consistency of model-structures such as depicted in Figure 21. Here we see a situation that a requirement model is satisfied by an architectural model which is refined into another architectural model.

4.8 The difficulty in maintaining traceability

Maintaining traceability is not a new subject in the software engineering world. It has been around for quite some time now, and multiple studies have been performed to investigate the problems that are encountered when maintaining it. To indicate the problems that can be encountered when establishing and maintaining traceability we will present an overview of two independent sources that investigated the problems and deficiencies in this area.

One of the earlier investigations of the problems in specifically the requirement traceability problems is the one carried out by Gotel and Finkelstein in 1994 [Gotel'94]. The authors present an empirical study, held among over 100 practitioners together with an evaluation of current support in the area of requirement traceability. One of the main underlying problems they identify is the lack of a common definition on requirement traceability. Gotel identified that the definitions mainly differ by the goal of the traceability. Definitions where either:

- Purpose driven, defining what the it should do;
- Solution driven, defining how it should do it;
- Information driven, emphasising the traceability information;
- Direction driven, emphasising the traceability direction.

The problem with these differences in definitions is that these definitions all have different scopes, and cannot be united to cover all concerns. This diversity in definitions on traceability makes it very hard to

maintain it correctly. If different people conceive the concept in different ways, then maintaining it with this diverse group of people will become impossible. There is a need for a uniform and consistent definition on this concept to achieve maintainability.

The authors also summarized the problems in requirement traceability as perceived by the diverse practitioners. We will mention these problems in a point by point manner:

- The granularity of the traceable entities is often too coarse grained;
- The integration technology is often too mature;
- Information needed for maintaining traceability is sometime hidden;
- Project longevity causes the traceability to deteriorate.

The authors furthermore have a unique approach in separating the traceability process into two main parts, being pre and post requirement specification. The authors identify that the main problem experienced with traceability can be attributed to a poor pre requirement traceability phase. They specifically mention a number of problems created by this phenomenon, grouped according to two types of stakeholders:

- Problems identified by providers of the information to create the traceability links:
  - Traceability is perceived as an optional extra. As a result not the appropriate resources are allocated;
  - Insufficient recognition by management on the amount of work it takes to maintain the traceability;
  - An Imbalance experienced between the amount of work involved and benefits perceived;
  - The Individual invested effort is ad hoc and localised. A combined & fulltime responsibility by all is needed;
  - A lack of agreement on end user requirement. A focus exists on immediate needs of the user;
  - The concerns of pre-requirement traceability tend to decrease after a sign of on the requirement document;
  - Information needed to create the links can not always be obtained;
  - Traceability is not maintained up-to-date;
  - Insufficient feedback on best practices in traceability and dedicated support.
- Problems identified by the end-users of the traceability links:
  - A Stereotypical end-user cannot be predefined. Their requirements differ greatly;
  - It is not always possible to define requirements in advance;
  - It is not always possible to define how access to information will be required;
  - Personal contact is needed because some information will be inaccessible / missed or out-of-date without it;
  - A great difference in the usage of the traceability depending on the circumstances per context. (traceability is dependent on the unique requirement on the traceability process)

Next to Gotel, also Knethen [Knethen'02] identified a number of deficiencies that can be identified in the current tracing approaches. We will give a summary of the problems that are identified in this work:

- No clear conceptual trace model is defined:
  In current approaches, mostly a conceptual model which defines the relations and entities needed to form a trace relation is not defined.
  - Kind and granularity of entities to be traced is unclear: In current approaches a conceptual model capturing the different kind of entities and the relations between them is missing. As a result often optimistic predictions or inconsistent implementations are made in case of a change. Another problem identified is that elements which are included are often somewhat coarse-grained. Approaches that do provide fine-grained elements to carry out detailed impact analysis for example often lack the ability to perform impact analysis on a system wide level.
  - Relationships to be traced not clear: A problem with current tracing approaches is that it is often not defined what relations must be traced to for a specific purpose. Another problem identified here is that knowledge about how and why documentation entities are related is often missing [Knethen'02]. A direct consequence from these problems is that traces may be incomplete due to the inadequate knowledge on the purpose of relations, or that some traces are even unneeded due to incorrect identification of relations. There are only a limited amount of approaches that support some kind of support on the semantics of relations in order to support the traceability process. That approaches that do exist have a coarse grained conceptual model and require the user to define this model and often does not consider the entire system. Often also only changing and removing of elements is considered in an impact

analysis. In the case of addition of an element, often not enough semantics on a relation is known to automatically relate this new element.

- Insufficient process support in capturing and analysing traces:
  Process support is an important element in correctly identifying the needed traces. Without process support there is more risk on a incorrect set of traces or misunderstanding of these traces.
  - Setting of relations: A consequence of the lack of semantics of relationships and entities to be traced is that it is often impossible to automatically set the traces to support a specific goal. As a result, the traces have to be set manually which is a labour intensive task. Automatic setting of traces can be achieved using name tracing (matching on naming), but is often more successful when the semantics of a relation are captured using typing of the relations. The authors mention that the automatic setting of traces using semantic knowledge through the types of relations is possible for representation relations, but more difficult for dependency and refinement relations because they are more dependent on the type of system.
  - Analysing of relations: Process support also assist stakeholders to use the information captured in the traces. An absence of this support hinders the stakeholders to use this information to achieve their goals.
- Tool support insufficient: Tool support assisting in the capturing and representation of trace links is currently available. Tools that however assist in automatic capture of traces, impact analysis or propagation of change on different abstraction levels are however not very much available. Currently most tools are highly human driven (and subjective) and are as a result very much dependent on the person using the tools. Stakeholder not directly seeing the benefits of capturing the trace information can hereby frustrate the traceability process.
  - Capturing of traces insufficient: Traces should be captured as the artefacts/elements are created. The capturing process should also be automated as much as possible
  - Analyzing of traces insufficient: Automatic identification of links between artefacts is currently often lacking.
  - Configuration management is difficult: The management of the artefacts like requirements is often restricted to a certain group of stakeholders to avoid inconsistencies. This is however a danger for the traceability process because certain stakeholders can not access the needed information to retrieve and alter the trace information. As a result, not all available information will be maintained.
  - Interoperability of tools: Tools supporting the development process and handling the artefacts of the process often do not interface with each others and dedicated traceability tools. As a result not all the information is available in one tool, which frustrates the process. One problem might be that information is not found, but it can also be possible that certain traces can simple not be created because an interface between tools is missing.

4.9 Summary

This chapter introduced the concepts of traceability by discussing existing definitions of traceability. We have discussed the importance of traceability and defined a number of reasons for the growing attentions in tracing. After having established the importance of maintaining traceability, we discussed the elements that comprise traceability. By discussing a survey on tracing approaches, we have identified the core concepts in a number of existing tracing approaches. We described the elements and relations that are using in current tracing approaches. The lack of consistency in defining the types of relation that exist, have lead us to redefine the types of relations that we use in this thesis. We have differentiated inter, intra and within-model relations.

We conclude that the there is no consensus on the meaning of tracing in literature, and that the concept is highly dependent on the goal of the stakeholders. Because of this lack of consensus we defined trace as a role of existing relations. This enables us to treat interlevel, intralevel and within-model relations as traces. In order to support this vision on tracing, we defined a metamodel for trace relations. This trace metamodel allows us to create intralevel, interlevel and within-model relations as traces. We defined two new relations to use for tracing purposes; satisfies and refines. We have formally defined these relations. After discussing the (formal) descriptions of these relations we defined the concept of consistency of a model. We use the existing formally defined relations to formally define the concept of consistency. This definition will later on be used to implement the consistency checking in tooling.

The chapter is concluded by identifying the problems in current tracing approaches, as identified in a number of literature resources.

5. Validation of the metamodels

After having discussed the metamodels for both requirement and architectural modelling, we have to show that these models can actually be used in practice. This means that these models need to be validated in practice. Otherwise these models could just as well be meaningless, as they are never tested on actual use in software development. Validation, in our case, means that we will take input from a software development project, and use this information to create concrete models based upon the metamodels. If the information can successfully be put into the models, effectively containing the same information as the original documents, this means that for this particular case they are validated. Because a full scale study, monitoring a running project, and collecting data over time, is out of the scope of thesis, we will only perform one mapping of a real life document onto both metamodels. We will first define what we mean with a mapping between documents / models.

We define a mapping as: *A mapping between two models establishes a 1:1 relation between each element in the source model and a conformant element in the target model. Conformance between elements is (in our case) established when two elements can carry the same information.*

We will first discuss the validation approach that we have used to validate the models in Section 5.1. Because the two metamodels are validated with different information from different projects, we will discuss these two separately in Sections 5.2 and 5.3. For both metamodels we will, shortly discus the input used to validate the metamodel, map the input model to our metamodels and finally present the results of this mapping in a conclusion.

5.1 Validation approach

Before diving into the actual validation of the models, we first have to establish whether this kind of validation is applicable, and will show that our metamodels are usable in practice. A number of approaches for validating claims in science papers are summarized in [Zelkowitz'07]. This paper gives a classification of validation methods, by offering a 14-scale taxonomy of validation techniques. This is a refinement of an earlier described taxonomy, also by Zelkowitz [Zelkowitz'98], in which a 11-scale taxonomy was used. The goal of the taxonomy is to enable researchers to determine the strength of their validation. Each validation technique, described in the work of Zelkowitz, includes their strength and weaknesses, enabling researchers to benchmark their validations. The framework differentiates strong and weak validation techniques. Strong validations include various empirical experiments. The weakest validation in the framework is No-experimentation indicating that validation had been appropriate, but was not present. In [Zelkowitz'98] [Zelkowitz'97], the validation methods are furthermore categorised by being either a observational, historical or controlled validation method. Observational method uses data from projects as they are being developed. Historical, uses colleted data from projects, already completed. Lastly, controlled methods provides for multiple instances of an observation for statistical validity [Zelkowitz'98].

Our way of validation the derived metamodels are difficult to classify. We use data, from completed projects, in which we will study the structure of the product. This is often called 'legacy data' validation. Here the legacy products of a completed project, called artefacts, are studied [Zelkowitz'98]. These artefacts usually include source code, specifications, design etc. Zelkowitz [Zelkowitz'98] also mentions that when only source and specification documents alone are considered, this is called static analysis. Here the completed project is examined, by looking at the structure of the product. This way of validation often uses the structure of the product to draw conclusions (like for example using the lines of code to try and measure complexity). We will however use the historical data to try and fit it to our derived metamodels. This fits more in the category of observational validation.

Because we lack the resources and time to study actual projects as they occur, we are bound to a smaller project which we will execute ourselves. We will map our derived metamodels on the historical data, to show that it can be achieved. We can however not call this a case study, because this requires the monitoring of a running project, and collecting data over time. This is outside the scope of this thesis. Instead our approach fits the description of a so called assertion. Here, the developers (we) are both experimenters and subject of study [Zelkowitz'98]. Zelkowitz mentions that this is often used as a preliminary test, before more formal validation will be caries out. This is true in our case. In the

work of Zelkowitz this kind of validation is found to be a weak kind of validation. The main reason for this is that; These kinds of validation are potentially biased, since the goal is not to understand the difference between two treatments, but to show that one particular treatment (the newly developed technology) is superior [Zelkowitz'98] . We acknowledge this fact, but it remains a valid validation technique, mainly used as a basis for future experimentations. Because this research covers the highly immature domain of tracing, we will accept, for now, a weaker form of validation. This can later on always be expanded to a full blown case study, fully validating our metamodels. It is however outside the scope of this thesis to achieve this. The kind of validation that we have chosen, meets our requirements; showing that for a representative document from the practice, it is possible to use our metamodels.

5.2 Validation of Requirements metamodel

This section will describe the validation of the requirements metamodel, using the requirements document, delivered by Chess. We will first shortly describe the requirements model. After this, we perform the actual validation.

5.2.1 Description of the requirements model.

Validating the requirements metamodel involves creating a concrete instance of the requirement metamodel based on a requirements document from a software development company. Chess, a software development company also shortly mentioned in the Project background, delivered us with a real life example of a requirement document, used in one of their development projects. The project involved creating a software product which was able to deliver medical information to a third party without revealing the privacy sensitive data on the patience. This privacy insensitive data could then later be used for statistical usages without informing the party creating the statistics about any personal data. This is achieved by substituting personal details with pseudonyms. The medical (privacy sensitive) information is supplied to the system in a file. This file needs to be processed by the system to remove any privacy related data, and should be delivered to another system. The system which receives this file can now perform statistical analysis on the medical data without that it known any personal data about the patients.

5.2.2 Conformance of the metamodel

In order to put the information contained in the requirement document into an instance of the metamodel, we first performed an intermediate step. This step involved (manually) transforming the full requirement document into an intermediate form. This intermediate document contains the same information as in the original requirement document, but is put into a big decomposition tree, as this is the main structure used in the original document. The original document was divided into several (sub) sections in order to follow a certain specification standard. This, mainly layout related information however makes the mapping to a requirement model more difficult. The intermediate document is a large decomposition of all requirements without any layout whatsoever. Next to this we also tagged every piece of information that was found in every requirement with the matching attribute or element in the requirement model that could contain it.

The reason for creating this intermediate document basically twofold; first it eases the mapping to the requirement model because it more closely resembles the requirement model. Because it already contains the tags to which element it should be mapped, it a labour intensive task of mapping each tagged-information to an element in the model.
The second reason for creating the intermediate form is that it allows us to perform analysis on the entire document without actually mapping it to the requirement document. By just analysing whether we can tag every piece of information to a element of the metamodel we have established whether all information can be represented in the requirement model. All that is left hereafter is the labour-intensive task of really mapping the intermediate model to the requirement model.

Because the entire requirement document contains a large number of requirements, we have chosen not to map each requirement document element to the actual requirement metamodel instance, but instead to first map every requirement document element to the intermediate form. After this we map a number of these intermediate form artefacts to the requirement model. We tried to find a requirement

for every mechanism in the requirement metamodel at least once. They other way around, if no attribute or element could be found in the metamodel to represent the information, this would mean a problem for the mapping, and this would result in a problem in the validation. We have chosen to map at least 20 and a maximum of 40 requirement element to the requirement model.

When the validation of the requirement document is successful, it should be possible to recreate the original document content, based on the information from our requirement model instance. This also validates the requirement document as the source of a requirement document-generator.

The document and the selection we took

The requirement document is split up in two parts, each describing a slightly different application part. Together these two systems form the complete system for the customer. Both of these application parts have a requirement section related to them. We have chosen to only take the biggest application part as a part of the validation. Because even this part contains over 60 requirements, we have taken a subset of this part of the requirement document. This of course cripples the validation, because only a part of the entire requirement document is considered. But as we mentioned before, the most time-consuming part of the validation is the actual inputting of the information into the requirements editor, and hereby creating the requirement model instance. This is however not the most important aspect of the validation. We chose to only really model the mentioned subpart of the requirements. The remaining of the requirements, we have however transformed into a intermediate form, tagging the information with attributes and elements following the structure of the metamodel. We scanned this intermediate document thoroughly for any problems that might arise when this intermediate model would have been put into an actual requirement model instance. Any problems or special situations will be reported here.

Mapping the document onto the requirement metamodel

Requirement name

One of the most common found elements in the requirement is a (rather large) requirement name which partly describes what the requirement does. It is often a simple text string which can be mapped to the name attribute of a requirement. Sometimes, the names are rather large, as often seen in document based models. It is sometimes needed to reduce the text used in the name, and instead place this text in the description attribute.

Requirement description

Most of the explanation of the requirements is done in a small section following the name of the requirement. This text is of most importance to the developer implementing a requirement, because it describes what should be implemented and in what way.

External Object

A big portion of the requirement document contains examples, explanations and images describing how certain requirements should be implemented or the way that certain functionality should be delivered. These are often included as tables, images or just example texts embedded in the requirement document. To be able to include these elements in our requirement model, we can use the external object element related to a certain requirement. Instead of actually including the table or image inside our model, we refer to a location which contains these descriptive elements as a file. When for example an image describes how a certain screen should look like, we can include an external-object which has the location of this image as an attribute. This enables us to include the information from the requirement document while not having to actually model texts and images inside our model. This mechanism is also suitable for generating a requirement document based upon this model. Al external object should then just be visited and parsed into a document based model.

Another element in the requirement document which needs to be represented in our model are the appendixes further describing how a certain requirement should be carried out. This includes an elaborate listing of steps and (technical) descriptions. One option would be to refer to these appendices just as in the original document, but now through the use of an external object. This keeps

the original model clean and compact, but also has a downside. The downside is that the individual information elements inside the appendixes, which can also be regarded as (sub) requirements, cannot be used for analysis or tracing. When it is useful and also desirable to do carry out analysis or tracing-steps on these (sub) requirements in the appendixes, then another approach would be to model them as a separate requirement model instance. This would in fact create a separate document (called then a model), just as in the original document the appendix could also be a separate document to which is referred. This separate requirement model describing a specific appendix must however also be linked to the requirement which contained the initial referred to that appendix. This can be achieved using the refines relation defined in our trace metamodel. We defined this relation as "a relation between two models or model elements which convert a more abstract model into a more concrete model". This relation could be used to transform a rather abstract requirement which needs additional explaining through an appendix into a model describing the (concrete) details of that requirement.

Stakeholders

Stakeholders are not mentioned explicitly per requirement. The document does however specify which parties are involved in which system. We have therefore mapped these parties to the (business) requirements of the (sub) system which they will be going to use.

Taxonomy

The document we studied did not contain any elaborate taxonomy of requirements. The only distinction that can be made is the difference between the goals of the system and the actual requirements. We have mapped the goal of the system to the business requirements, and mapped all other requirements to software requirements. There is furthermore no difference between functional or non functional requirements.

Another aspect which isn't mentioned specifically in the document are constraints. We however discovered that some of the requirements mention them explicitly. An example of this is: "If there is a failure sending the file, then the zipfile must be written in the directory of the inputfile". This textual description is now just a part of a requirement text. We however interpreted this sentence as being a constraint with a mandatory nature.

Relations

The relations found in the requirement model are very limited, and basically mainly consist out of a decomposition of requirements into sub requirements. Mostly this decomposition is indicated by using bullets, indicating sub requirements and adding details to a requirement. These sub requirements are often just mentioned in the describing text of a requirement, but can be distilled in this way.

Decomposition is also found in the numbering of the sections of the document. 3.2.1 for example describes the PVM application part, and 3.2.1.1 describes its purpose and 3.2.1.2 describes its start-up requirements. Sometimes the decomposition is not directly clear, but a translation can be made however.

Another construction which is used in the requirement document is the summation of a number of steps that need to be performed. We translated this to a decomposition which has the steps to be performed as mandatory children (AND-decomposition). Some of these steps dictate that other steps must be skipped if one step is performed. We can translate this to either a constraint which excludes other steps (modelled as requirements), or can just mention this in the description.

Other relationships found in our requirement model are not found in the requirement document.

5.2.3 Results of the validation

After performing the validation of the requirement document in the way we described above, we will now present a summary of the elements found in the requirement document and its matching elements in our requirement metamodel. For some elements in our metamodel we didn't find any elements in the requirement document. The results are presented in Table 1.

Element in the requirement document	Element in our metamodel
Requirement section - Section Name - Section content, describing the requirements	Requirement element - Requirement name attribute - Requirement description attribute
- Images / Tables / Examples describing the requirement.	- External object containing references to the files containing the Images / Tables / Examples
- Appendixes containing details of a certain requirement	- External Object containing the location of the appendix. Or a separate requirement document, related by refinement to the supported requirement.
List of terms Containing involved parties - Involved parties - List of other abbreviations	Stakeholder element - Stakeholders - Added to the description where they are used.
Taxonomy - Goals of the system - Requirements - Computer resource requirements	Requirement types - Business requirement - Software requirement - Non-Functional Requirement
Relationships - Main structure o Bullets containing per requirement details o Section numberings - Exclusion of one when implementing another	Requirement relations - Decomposition relation, AND type - Dependency relation, Excludes Type

Table 1: Requirements validation summary

After finishing the mapping of the requirement model in the case study to the intermediate model, and also mapping parts of this model to an instance of the requirements metamodel, we can conclude that a mapping is feasible. We conclude this from the fact that every model in the source requirement model can be mapped to elements in the target requirements model. It should be possible to recreate the original requirements document based on the information stored in the requirements model. Apart from the layout related details, all the information is available in the requirements model. These layout details are however usually captured separately, and can be provided by for example templates for the specific requirement document.

5.3 Validation of architectural metamodel

For validating our Architectural metamodel we took a different case-study that the one used in our validation of the requirements metamodel. The reason for this is that the case-study of the ZorgTTP project did not include an architectural design. Other projects within Chess that do include an architectural design are often of such a scale, that they are not practice to be used as a case study for us. We have therefore chosen to select an architectural design used in the work of Joost Noppen [Noppen'07], in which he describes a high level architecture for the Traffic Management System. We will first quickly describe the setting for case study, after which we map the elements in the case study to elements in our metamodel. The latest section will cover the conclusions on the validation step.

5.3.1 Description of the architectural design model

The Traffic Management System (TMS) is a regulation system, designed to monitor and regulate the traffic flow on a national scale [Noppen'07]. The architecture that is described in the work by Noppen should offer an architectural design for a new TMS which is being developed. The system should provide technical support for monitoring, controlling, managing, securing and optimizing the traffic flow effectively [Noppen'07]. The case study does however not involve the analysis of the entire TMS as it is described above. Instead the case study involves the task allocation part of the TMS, which handles its allocation based on scenarios and available traffic information. A description of this part of the TMS, according to the stakeholders, as written down in [Noppen'07] is as follows:

"The TMS should provide assistance when the traffic flow is limited. It is the job of the TMS to support operators to coordinate the activities that should reset the traffic flow to its "normal" state. To achieve this, the TMS should support the action coordination for traffic flow normalization. This is done by allocating tasks and scenarios to system operators. The Task Allocation part should gather and store information about traffic in its direct and indirect geographical vicinity. To communicate the tasks and actions, the TMS should be able to access its connected roadside systems. In addition, the TMS should support systems operators in identifying tasks and actions that will normalize traffic flow as fast as possible."

An overview of the most important requirements of this part of the TMS can be found in [Appendix A]. Here the requirements of this part of the TMS are modelled in our requirements metamodel as described in earlier chapters.

The Architectural model

The high level architecture of the Traffic Management System is depicted in Figure 22.This part of the architecture is aimed at task allocation and communication with the outside world.

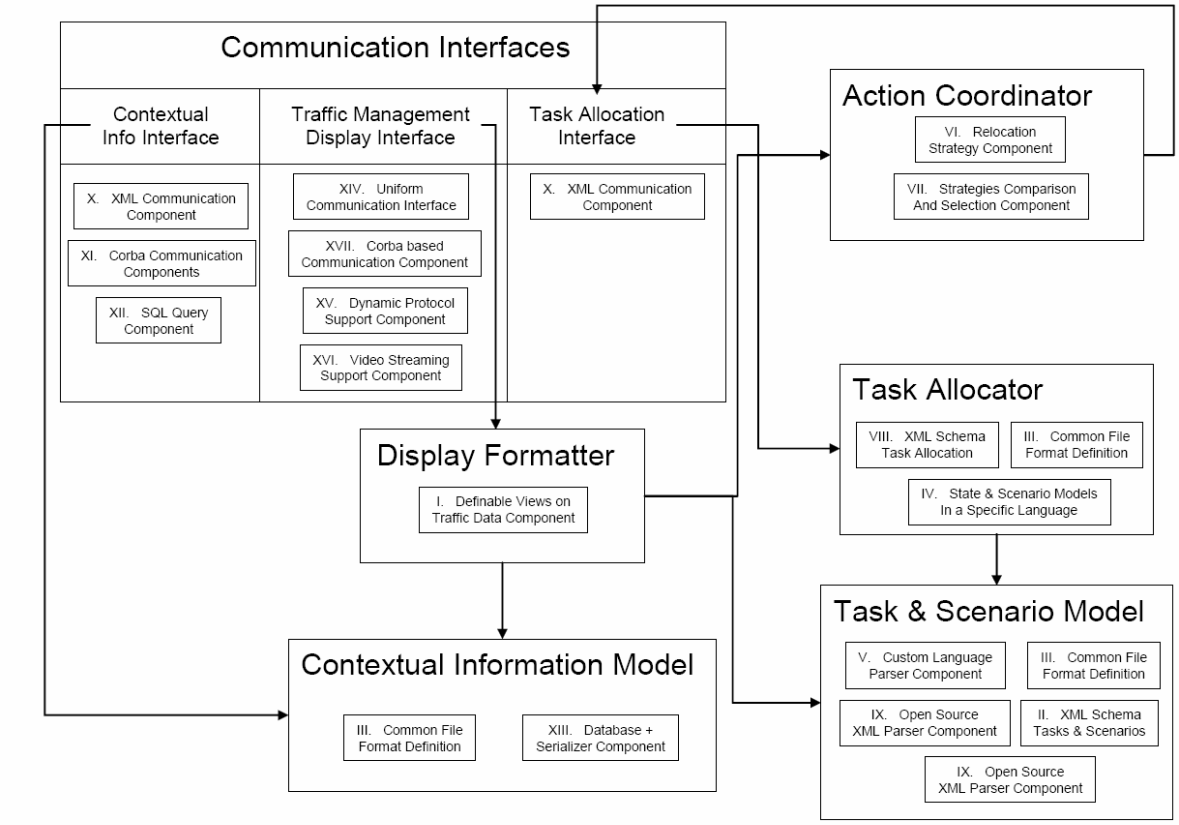


Figure 22: TMS high level Architecture

The large blocks in the figure represent abstract entities which contain the most important functions of the system. They really represents unit of implementation, sometimes to be developer or sometimes to be used as a COTS component.

The arrows between the boxes represent a usage relation between two abstract entities [Noppen'07]. The action coordinator for example communicates with the Taks & Scenario Model to be able to describe the actions in terms of tasks and scenario's. According to the work by Noppen, the grouping of boxes in the interfaces is just to improve the readability of the diagram. It is not meant to describe decomposition or other kind of construction. The smaller boxes inside the big interface box really imply individual interfaces. The fact that they are composed together in a single box, just indicate that when referring to this containing box, one really refers to all the containing interfaces separately.

5.3.2 Conformance of the metamodel

We will now map the information contained in the High level architecture of the TMS onto elements of our Architectural metamodel. We will follow a similar way of working as done in the requirements metamodel validation, discussing the mapping per element in the TMS architecture.

Conceptual block

The conceptual blocks in the diagram really represent implementable functional units. These can easily be mapped onto the component element in our metamodel. As we can see from the architecture of the TMS, no additional attributed are specified per implementation unit, apart from its name. The names in the architecture sometimes tend to be quite large, almost hinting at an explanation of what the block does. In this case the description attribute of the metamodel would be a good place to store this information.

Sub block

What we see in the TMS architecture is that implementation units are often split up in smaller sub units (or –blocks), to add the level of detail. Often the implementation unit uses a number of smaller implementation units that together provide the functionality of the surrounding implementation unit. In this case, it is really a decomposition of components into smaller components. We can represent these block/sub block relation in our metamodel by using the Module decomposition-connector. The sub blocks are in this case just modelled as normal components, which are connected to the containing block, using the decomposition connector.

Connection between the blocks

As indicated by [Noppen'07], the arrows indicate a usage relation. This means that they use each others functionality to be able to achieve their own functionality. The architectural metamodel offers a special type of connector for this kind of relation between components. The ModuleConnector is especially designed to model a relation between static components (as apposed to dynamic, runtime processes), such as the implementation units of the TMS system. The Module connector has a number of types, of which we already saw the decomposition type in the block/sub block construction. To model the usage relation, the Module uses-connector is the most suited. This is especially aimed at modelling a relation between static components that needs each others functions in order to be fully functional.

Interfaces

Another element found in the architecture of the TMS are interfaces of components. These are grouped together in one big element describing all the interfaces in the architecture. As indicated by [Noppen'07], this is to improve readability, and does not hint at any decomposition of other structure amongst these interfaces. It must however be noticed that the links to the grouping mechanisms are also established. After reading the explanation by [Noppen'07], we conclude that when a relation refers to a group of interfaces, we can interpret this as multiple links being established to each individual interface. The interface element which is available in our metamodel can be used to represent the interfaces found in the architecture of the TMS. Because the interfaces of the TMS are high level interfaces, not describing methods or functions, the operation attribute must be used to represent the name of the interface. The type attribute is in this case not used. The interface elements will in this case be connected to the component elements, representing the functional units. Components can furthermore have multiple interfaces, and interfaces can be used multiple times by more that one component. This offers a way to model the reuse of the interface as for example the task allocation interface which is referred to multiple times.

5.3.3 Results of the validation

After performing the mapping between the high level architecture of the TMS and our metamodel we again present our summary of this validation in a tabular form. The result can be found in Table 2.

Element in the TMS architecture	Element in our metamodel
Conceptual block - Name	Component - Name attribute
Sub block	Component - Connected to the containing block by a Module Decomposition-connector
Connection between the blocks	Module Connector - Uses type
Interfaces - Grouping of interfaces	Interface - Connected to the elements which use it - When connected to a group of interfaces, include multiple connections to each separate interface in the group.

Table 2: Architectural validation summary

Having mapped the components of the high level architecture of the TMS on our architectural metamodel, we can conclude that we did not come across any mapping problems. We can conclude that the metamodel is valid for this particular case.

5.4 Summary

Two examples, taken from the software industry practice, have illustrated the validity of both the requirements and architectural metamodels. Before we validated the models, we have first described the validation technique that we used. Due to time constraints and limitations in resources we performed a validation classified as assertion. In literature describing validation techniques, this validation technique is coined rather weak. The reason for this is that the experimenters are both experimenters and subject of study. Future research could use a more extensive validation technique to fully validate the metamodels.

The requirement metamodel is validated using a requirements document of a past software project, developed by Chess. Because this document is a too large to directly validate onto our metamodel, we have first used an intermediate model. We mapped a complete section of the original document onto this intermediate model. We have mapped every element type in the intermediate model onto the actual requirement metamodel (minimal 40 requirements). We have shown that every element in the section of the original document can be mapped onto the intermediate model. Furthermore we have shown that every kind of element in the intermediate model can be mapped to our requirement metamodel. We conclude that every element in the original requirement document can be mapped onto an element of our requirement metamodel.

For the architectural metamodel we have taken a different approach, directly mapping an existing design document onto our metamodel. The design document is taken from a PhD thesis covering the software design process. We again performed the mapping and concluded that every element of the original design document could be mapped onto our architectural metamodel.

6. Tool support

In this chapter we will discuss the current tooling that has been designed to support the management of traceability. This tool support will focus on improving the consistency between requirement and architectural design. The tool support relies on a number of frameworks in order to provide the user with editors for the input. We will first shortly discuss these frameworks and their relation to Model Driven Engineering in the Sections 6.1 and 6.2. We will discuss the general architecture of the tooling in Section 6.3. We will describe the actual tooling itself, as it is currently provided, in Section 6.4. In Section 6.5, we will discuss the pros and cons of the current tooling in which a critical look is taken at its use in practice. Section 6.6, is dedicated to describing possible extensions to the tooling and future work to be done in the direction of building a complete toolkit to support the entire traceability process.

6.1 The Eclipse Rich Client Platform

A big disadvantage in current tools assisting in the maintenance of traceability between design artefacts is that they are often separated from the actual development tools. As a result, the process of maintaining traceability is often separated from the development process. It is often the responsibility of a person who also has other (development) tasks. When traceability is maintained in this way, there is great risk involved that traceability is not given enough attention, and that the traces erodes over time. A first step to improve this situation is to integrate the tools supporting in the traceability process with the development tools themselves. In order to achieve this, a logical step would be to integrate the tool support within the Integrated Development Environment (IDE). When traceability support is integrated in the IDE, it offers the possibility to reuse artefacts created in the IDE like for example code artefacts.

The Eclipse project

One of the most popular IDE, for multiple programming languages is the Eclipse IDE. One of the reasons for its popularity is that it is highly extendible. Originally the project was started by IBM in the mid to late 90's, to create a common platform for all IBM development products. An advantage of this approach is the compatibility of the tools being built on this common platform. The first product from the Eclipse project was a IDE for the Java language, which was built together with the generic platform on which it relied.

In order to create more tools based upon this common platform, IBM started the so-called Eclipse consortium. Initial adopters included the major partners of IBM, Rational Software, TogetherSoft and a few of the competitors of IBM.

The generic platform beneath the Java IDE has proven to become increasingly important, when we look at the history of the Eclipse project. This soon offered a basis for many tools, ranging from IDE's for Java to C++, PHP and Python. These tools can be integrated and combined as needed, all within an Eclipse application. The main technology for enabling this is the underlying, Eclipse Rich Client Platform (RCP). This RCP is now completely separated from any IDE related components and offers a very general basic platform for application development.

The Eclipse RCP

The RCP is a generic platform for running applications. Basically, the Eclipse IDE is just one of these (highly sophisticated) applications running on top of the RCP. A RCP application is in reality a collection of smaller components, called plug-ins, which represent the basic unit of functionality within Eclipse. An RCP application is then a collection of these plug-ins together with a runtime specification on which they run. Even the RCP framework itself is a set of these plug-ins, bundled above the Java Runtime Environment (JRE). We can graphically depict this architecture in Figure 23. The Eclipse RCP is the basis for the applications above it. The RCP framework itself is only dependent on the basic subset of the Java runtime (Foundation). Plug-ins cooperate by using each others functionality. The IDE is for example dependent on the basic RCP, but also on the JDT (Java Development Tools, offering the Java IDE) and PDE (Plug-in development Environment). We will not dive into the details of

all the different plug-in in the RCP, as it is outside the scope of this thesis. It is however important to understand the notion of cooperating plug-ins when building RCP applications.

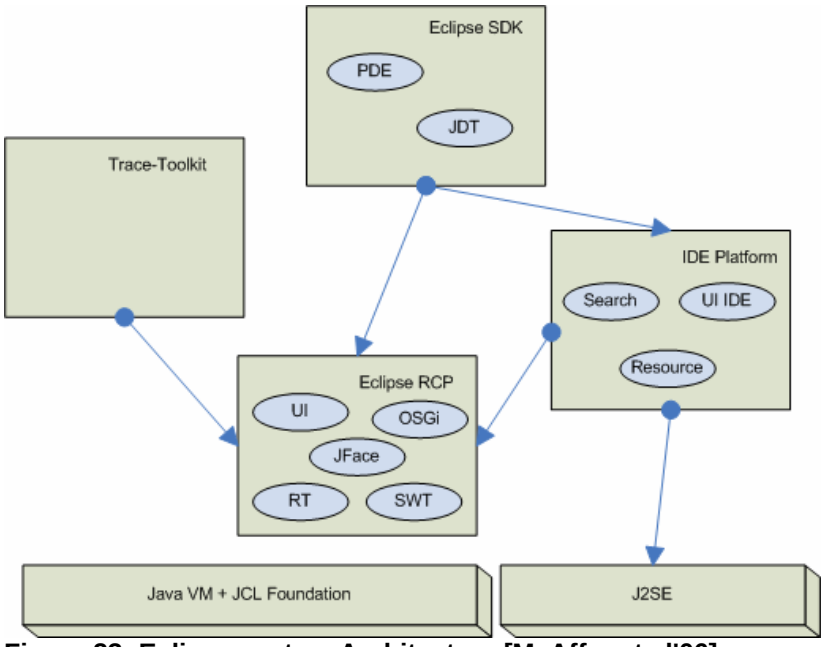


Figure 23: Eclipse system Architecture [McAffer et al'06]

Plug-in development

The great advantage of this flexible, plug-in (or component) architecture is that the Eclipse applications just a collection of plug-ins that can easily be extended. This offers the possibility of creating custom-built tools to extend the Eclipse IDE. We can treat the tracing tools as just another plug-in which can be integrated into the Eclipse IDE. This offers developers on a project team the possibility to maintain the traceability inside the development environment they use every day. In this thesis we only focus on the consistency between requirements and (architectural) design level. But we can easily imaging this being extended to lower level design artefacts like code artefacts.

6.2 The use of model based frameworks

Until now, we have devised a number of metamodels to enable the modelling of requirements, architectural and trace information. This offers the structure for storing the information that is available in both the requirements and architectural phases, and to relate this information in a separate trace model. This information is saved as a model instance which conforms to the discussed metamodels.

The model instances of the metamodels do however not need to resemble the looks of the metamodel (resembling the UML look). A requirement model can for instance be just a textual document, in which a mapping is possible from every element of the textual document to an instance of the metamodel. The most important thing is that we ultimately arrive at a model instance conforming to the metamodel. The representation of a model does become important when a user wants to put these models into a software system. In order to input the requirements and architectural information into a software system that can perform operations on it, a (graphical) editor or GUI has to be available. Building a GUI for a software system is often a work intensive and error prone task. It is nevertheless an important factor in order for a model to be really used. If a model cannot be input into a system in a convenient manner, then the user will eventually not use the models as input for the system. The choice for a specific representation to use in the tooling is a difficult one, and not specifically the focus in this thesis. We have already seen a number of possible representations in the earlier chapters, as described by Wieringa [Wieringa'05]. One convenient representation to implement is a matrix displaying two models on both axes. This approach is taken by a number of (commercial) companies developing traceability-tools, like for example IBM requisite pro and Telelogic Doors. We will follow this approach, and use a traceability matrix to establish relations amongst model-elements.

6.2.1 Model Driven Development with Eclipse

A solution to the (labour intensive) manual building of model editors is the generation of editors based on the provided metamodels. Following the MDA-based approach to transform models into (partially) other (more specific) models, we could eventual generate source code (low-level models) out of these metamodels. A project within Eclipse, called the EMF framework is a first step in this direction which enables users to generate source code and editors based upon metamodels.

6.2.2 EMF

The Eclipse Modelling Framework (EMF) is a project that recently gotten great attention because of its model driven nature and ability to generate code out of models. This framework is created with model driven development in mind, having models as input. EMF is a Java framework and code generation facility for building tools and applications based on structured models [EMF'08]. EMF offers an extension to the Eclipse platform by offering a way to turn models into customizable Java code. Models are seen as input to generate (part of) applications, instead of a way to describe or document a system. EMF offers a metamodel called Ecore to describe the data models. These models, which are instanced of the Ecore model, are called core or domain models. Ecore basically is a subset of UML (a subset of the class diagrams, specifically) that allows for the modelling of packages, classes, attributes and relations. These basic constructs offer enough constructs for us to model our metamodels and to generate standard Java code out of it.

These core models are saved in XMI (XML Metadata Interchange), but can be created in different ways like for example annotating java interfaces, exporting them from Rational rose, or using a graphical Ecore editor. Once the model is specified in Ecore, the EMF framework offers a generator to generate Java classes out of these models. These classes can be extended to own use, and the generator will be able to cope with this when regenerating. EMF offers more than this basic advantage of development time reduction through code generation. It also offers model change notification, XML (eXtensible Markup Language) based serialisation of the models, validation of the input model and a API for manipulating the EMF objects using introspection. EMF basically consists of two frameworks; the core framework and EMF.edit:

- *Core EMF:* The core EMF offers the ability to generate the Java classes from the Ecore domain model. This is the most basic functionality that can be offered by a generation framework;

- *EMF.edit:* The EMF.edit extends the functionality of core EMF by adding adapter classes that enable viewing and editing of the models. More importantly, this also offers a possibility to generate a fully functionally tree based editor to edit the models. This editor is a default implementation to show the functionality of the EMF.edit framework like adapters and command based editing. It possible to modify the behaviour of this editor to better suit the specific domain.

6.2.3 GMF

The Graphical Modelling Framework (GMF) is a graphical extension to the work delivered by the EMF framework. EMF basically offers the generation of model source code and the code for modifying the values of the model elements. As said, also a default treelike editor implementation is generated. Often, this treelike editing is not very well suited for every particular domain. A graphical editor is often better suited to edit the information in models. When we for example look for at architectural design modelling, we can easily imaging that this is not a very ideal domain for treelike editors. Instead it is more natural to draw the architectural components and relations amongst graphically.

The GMF framework is designed to provide automated support for generating graphical model editors, based on the same input model as EMF. GMF is therefore highly relying on the functionality of EMF. GMF is also highly model driven, as a graphical editor can be generated almost solely by giving models as input to the framework. GMF achieves this functionality of generating fully operational diagram editors by bridging the gap between EMF and the Graphical Editing Framework (GEF). GEF is a framework designed to building graphical editors for the Eclipse platform. GEF uses native widgets normally found in the Eclipse platform, and is highly platform independent.

GMF offers a way to generate a GEF-based graphical editor based on the models provided by EMF. Both the editor generated by the EMF and GMF frameworks save the actual model information (the content of the model, inputted by the user), in a XMI file which is called the semantic data of that model. GMF however also has another XMI file associated with each model, in which the graphical representation data is saved. These two are separated because otherwise, GMF would 'pollute' the basic semantic data with graphical information. This is not desirable, because it would render the file unreadable for the other editors (like the tree based editor of EMF). References from the notation model to the semantic model keep the information in sync.

The input of the GMF framework consists of three models:

- The Ecore model, containing the model information. This is the same model as the one used in the EMF framework.
- A graphical definition, defining how each element defined in the EMF core model looks like graphically (e.g. a rectangle, circle etc).
- A Tooling definition, defining how the menu's to edit the element instances, look like.

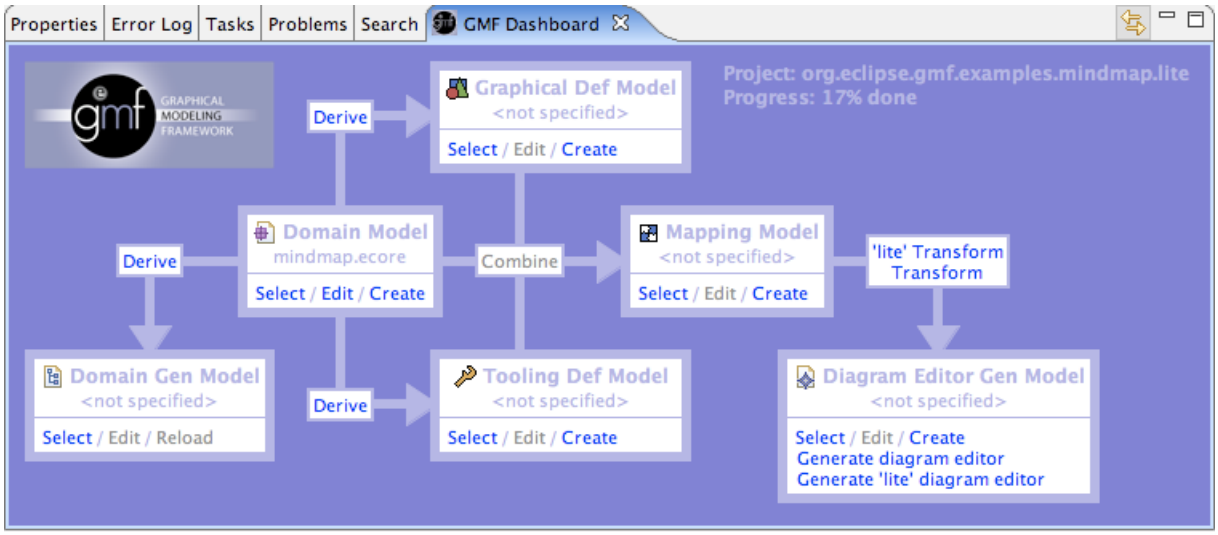


Figure 24: GMF Overview

We can see all these models back in the overview picture, in Figure 24. We see that the above-mentioned models are linked together in a so-called mapping model, which bind the Ecore model elements to a graphical representation and a tool to edit them. Once the mappings for each element are made, this mapping model contains a generator to generate an Editor generator. This last generator is finally responsible for generating the editor itself. Some options for specific generation are given, like for example generating a standalone editor, or a editor which is dependent on the Eclipse IDE plug-ins. It is possible to adapt the GMF generated code (as in the EMF framework), to the users specific wishes. This enables the creation of specific graphical editors for a specific domain.

One aspect of the overview picture that we haven't mentioned yet is the so called Domain generator model. This model is actually not a part of the GMF, but is the generator responsible for generating the basic java classes and editing source code to enable the GMF editors to 'talk' to the model classes. The content that is generated here is the same as the content generated for the standard EMF-tree editor we mentioned earlier.

6.3 The Architecture of the Tracing Toolkit

Now that we have discussed the possibilities within the Eclipse framework to (partially) generate editors based on the input of the (meta) models, we will now discuss how to use these editors to our advantage. What we want to achieve is one so-called 'toolkit' to be able to input both requirement models, and architectural models, relate these two and use these relations (traces) to keep the two consistent. In order to circumvent the labour intensive building of graphical editors, we can now, build these editors using the EMF and GMF frameworks. This gives us the ability to input the requirements

and architectural models into the system. We hereby also have the basic mechanisms in place to access and edit the model information ourselves, delivered by the EMF.edit framework.

We can now describe the approach we have taken to create the toolkit that we want. The general architectural overview of this toolkit is depicted in Figure 25.

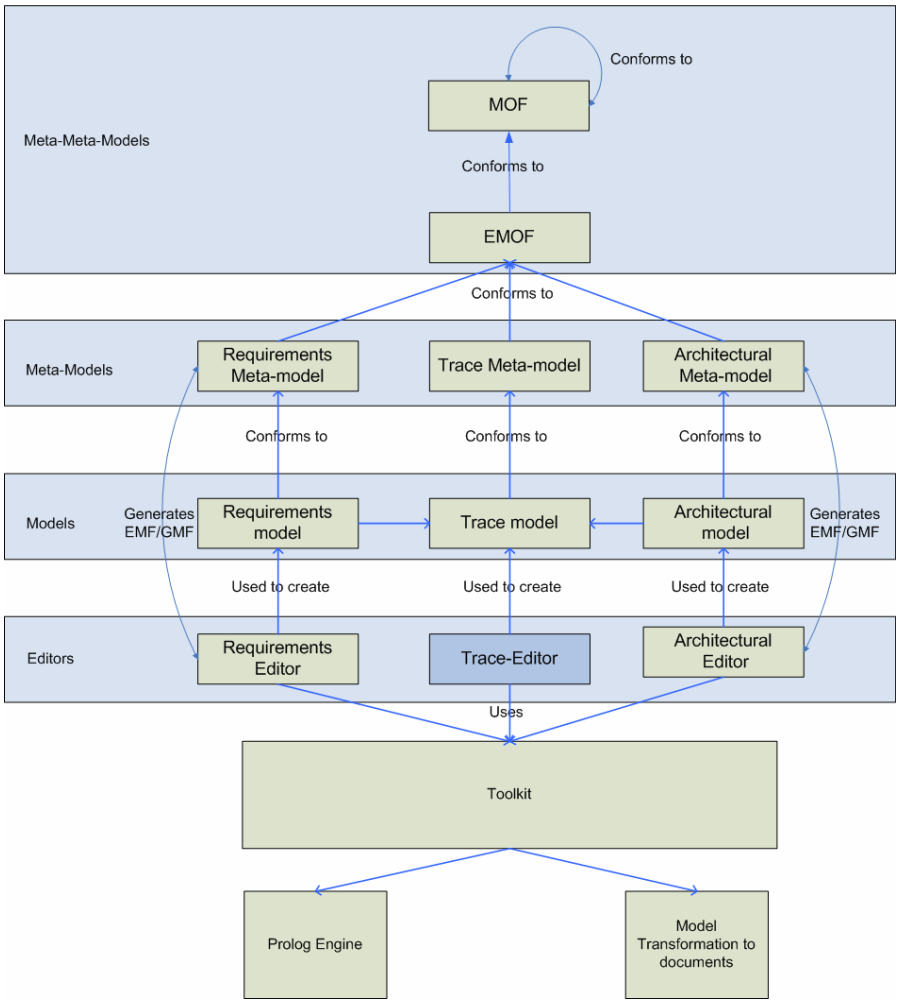


Figure 25: Tracing Toolkit architecture

6.3.1 Top level description

On the top level of this architecture we find MOF, which is a standard for model driven engineering, delivered by the Object Management Group. We will first explain the main ideas and concepts behind OMG's MOF in order to fully understand the layered architecture that we have used in our toolkit.

MOF

MOF is a four layered modelling architecture that provides a meta-metamodel as a top layer of this architecture, as seen in Figure 26. The OMG has subdivided the architecture in a number of levels, in which the lop level is called the so called M3-level. In this architecture, each level conforms to (or is an instance of) the layer directly above it. The top level meta-metamodel is an official standard, and originated from a need for a metamodel for the UML language. The UML metamodel is as a result an instance of MOF. The big question which automatically arises is, in which language is MOF then expressed? Or put it another way, of what model is MOF an instance? The answer to this question is, maybe not surprisingly: of MOF itself. MOF is hereby a so called closed metamodelling architecture, which means that it conforms to itself.

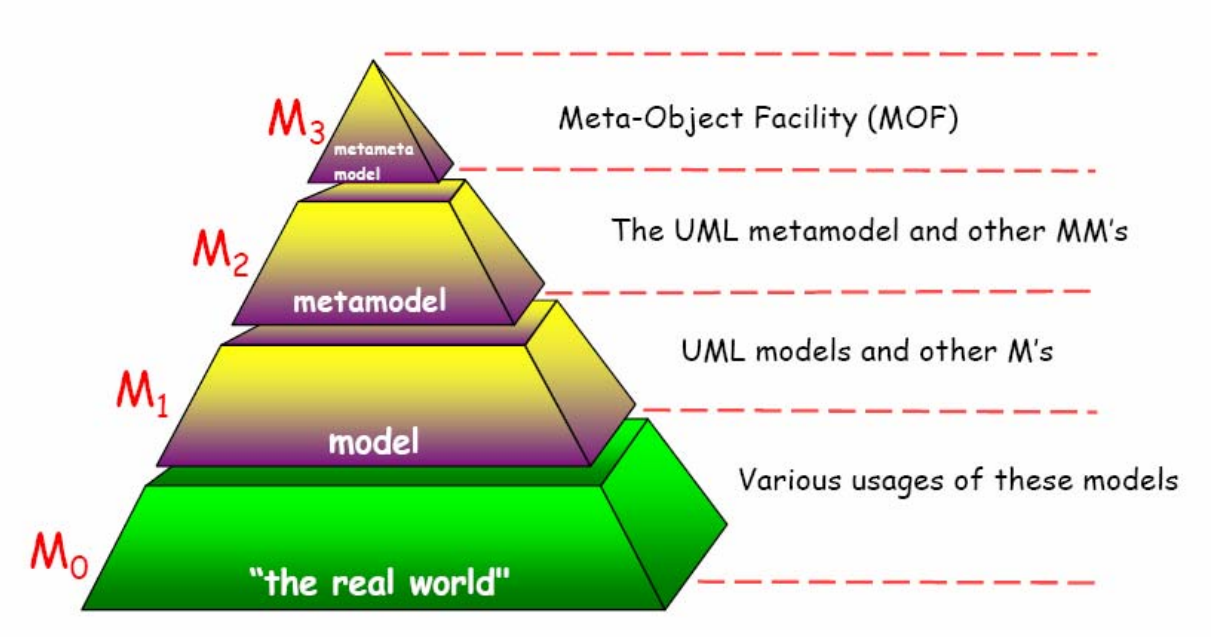


Figure 26: OMG Metalevels: MOF.

The layers below M3 will become more concrete towards the M0 level. The M2 levels of the MOF architecture contain metamodels. The most well-known example of such a M2 level model is the UML metamodel which describe the elements one can use in for example a UML class model. The models defined on the M1 level are more common to most software engineers, as these are the model instances as we known them. This could for example be a specific class diagram for a system under design. If we would go one level deeper, we arrive at level M0, being the actual real world things that are modelled at level M1. If we use the same example as before, M0 level models would be the actual classes which conform to the class diagram at level M1.

To clarify this system, a very well known analogy to MOF can be made to EBNF (Extended Backus Naur Form) which is used to describe lineages. This EBNF plays more or less the same role as MOF. It offers a top level metamodel (at M3 level) to describe a language which is also an instance of itself. In a similar way that UML metamodel is a instance of MOF, the Java grammar is an instance of EBNF. The Java grammar is then defined at level M2. One level below this would be an actual Java program, defined at level M1, conforming to the Java grammar. This is similar to a UML class model conforming to the UML metamodel. Level M0 would then finally contain the Java object in the Virtual machine. An overview of the OMG Metalevels is given in Table 3.

Level	Description	Example
M3: Meta-Metamodel	Used to specify Meta-Metamodels. Is an instance of itself	MOF, EBNF
M2: Metamodel	Used to specify Metamodels. Is an instance of a Meta-Metamodel.	UML Metamodel, Java Grammar
M1: Model	Used to specify the real world. Is an instance of a Metamodel	UML Model, Java Class
M0: The real world	Define the real world. Instances of an Model	An UML model instance, Runtime java Object

Table 3: OMG Meta Levels

EMOF and Ecore

What we see in the architecture picture of the Tracing Toolkit is that we encounter another model on the meta-metamodelling level called Essential MOF (EMOF). EMOF is basically a variant (or subset) of the complete MOF meta-metamodel. EMOF is actually a part of the MOF 2.0 specification that is used for defining simple metamodels using object oriented concepts. They are often used when the model instances need to be mapped to implementations such as XMI. EMOF offers next to this the ability for discovering, manipulating, identifying, and extending metadata.

One of the most widely known implementations of MOF is the aforementioned Ecore within the Eclipse project. Ecore is compatible with MOF 1.0 and allows the importing of EMOF metamodels. Because we have chosen to use the EMF framework in order to generate a lot of the editing code, we will use Ecore as a basic meta-metamodelling language. Ecore complies to the MOF specification as also seen in our architectural overview.

6.3.2 (Meta) modelling level and editors

The level below the meta-metamodel ECore, are the actual metamodels that conform to Ecore. We have specifically provided a requirement and architectural metamodel to express both requirements and architectural design as models. As we can see in Figure 25, these models are instantiated by real requirement and architectural models, and can also provide the basic editors for these models, as explained in the previous section.

Next to the requirements and architectural models, we also have to be able to instantiate models conforming to our discussed trace metamodel. The instances of this metamodel represent the trace relations that we can establish between the existing model elements. As for the requirement and architectural model, we also have to have a (graphical) editor to modify these trace models in the toolkit. We will, as mentioned earlier, represent these traces as a matrix with the elements of two models mapped on the axis. Each entry in the matrix will represent a trace between two model elements. The models that are represented on the axis of the matrix will, in our case, be restricted to requirements and architectural models. We will also restrict the types of traces that can be created to those that we have formally defined before; the satisfies and refines traces.

As we have seen in the trace metamodel, we will create source and target artefacts for every relation that is created. These artefacts will contain a reference to the actual model instances (e.g. a requirement model), which contain the referred elements. In this way, we keep the model instances separated from each other (loose coupling), keeping the trace model focussed on the actual information of the traces.

6.4 Fitting it all together.

Once we have the basic editors to edit the model instances of the metamodels we are ready to incorporate these in a toolkit to maintain consistency between them. We will refer to this tool as the Tracing Toolkit. We will describe the elements of the Tracing Toolkit in this section.

Based upon the problem statement of this thesis, we can derive a number of requirements for the development of our tool support. The complete list of requirements that is derived can be found in [Appendix C. Requirements Tracing Toolkit]. The top-level requirements can be summarized as follows: The system shall enable the user to:

- use the system within an Eclipse IDE;
- enter requirement and (architectural) model into the system;
- create (trace) relations between model elements ;
- store the model information in a repository for every model instance;
- perform consistency checking / trace analysis in order to maintain consistency between requirements and (architectural) design.

We will discuss how these requirements are achieved by discussing per requirement how they are implemented in the Tracing Toolkit.

6.4.1 Integration into the Eclipse IDE

The Tracing Toolkit is developed as a Eclipse RCP application. This means that the toolkit can be executed as a standalone application by being bundled with all its required plug-ins (EMF, GMF, PDE etc). This standalone application is a collection of cooperating plug-ins combined with a runtime that binds it to the Eclipse RCP application framework. By doing so, the Eclipse RCP will provide a platform to run our tooling on. The great advantage of building the tool as a Eclipse RCP is that the

internal functioning of the tooling is provided by a number of separate, cooperating plug-ins. These can be run as a standalone application, using the RCP framework, but can also be bundled for use in the Eclipse IDE. In this case, the application will just add behaviour to the existing (Java) IDE. When run as an IDE plug-in, the trace tool will offer the options to edit files associated with requirements, architectural design and trace models. These can now be opened within the IDE and will present the user with graphical editors for each of the models. We will discuss these editors later-on. The Tracing Toolkit consists of the following plug-ins:

- SimpleArchitecturalEditor: This plug-in offers the functionality to edit the architectural models in the tool. To achieve this, it offers a graphical editor to draw architectural designs. This plug-in is created using GMF
- SimpleRequirementsEditor: This plug-in offers the functionality to edit the requirements models in the tool. This is again a graphical editor to define the requirements, and interconnect them with relations. This plug-in is created using GMF
- SimpleTracingEditor: This plug-in delivers the basic java code, or API, to modify trace models using a generated framework. This plug-in is used by the Trace editor to instantiate and modify trace models. It is created using the EMF framework, without generating an default (tree view) editor.
- SimpleTracingUtils: This plug-in provides general utilities to perform the following tasks:
  - create Prolog predicates (discussed later-on);
  - assign unique identifiers to each model element;
  - perform string manipulations;
  - select which traces to use in the analysis of models;

  These functions are separated in a utility plug-in to allow all the other plug-ins to use it.
- SimpleTracingToolkit: This plug-in combined all the other plug-ins into a RCP application. The most important part of this plug-in is the trace editor, which delivers a editor for the trace models, using the SimpleTracingEditor EMF code. This editor allows the traces to be created between the other editors.

6.4.2 Editors for requirements and architectural design models

Both the editors for the requirements and architectural models are created using the GMF framework. The result is that the editors also resemble the look and feel of the default GMF editors. While the look of the GMF generated can be customized by the user, this is still a highly labour intensive task. While it is not our task to create fancy graphical editors, but instead to just offer the basic possibilities to editor our models, we have chosen not to do this.

Both the requirements and architectural editors represent the entities found in our metamodels as boxes and lines between them. We have chosen to model all (within-model) relations as lines, connecting entities. Other elements will be modelled as rectangles. A decomposition of requirements into two sub requirements will as a result be resembled as a requirement entity connected to the two sub requirements by two lines. Similar behaviour is found in the architectural editor, where architectural elements (components or processes) , presented as rectangles are connected with lines, representing the connectors.

While we changed the look and feel of the editors as little as possible, we did however change the behaviour of the editors. In order to achieve our goal of consistency checking, the editors give feedback on its consistency using the editors. The following behaviour is added to both the requirement and (architectural) design editor:

- Consistency checking upon opening a model: On opening either a model, the trace tool checks for every model element, whether it is consistent with respect to other models, using the traces that are selected. How traces are selected in the tool will be described later-on. As described in section 4.6. we check this consistency using the satisfies relation between model elements. How this is achieved will be described shortly hereafter. Elements that are satisfied are displayed in a green colour, and a ‘satisfied’ text is added to the element. When an satisfied element is selected, the properties of the element will display by what element the requirement is satisfied.
- Maintaining consistency upon editing: When an element in the editor is edited, the tool will again use the selected traces to check whether this element still is satisfied, and as a result will

compromise the consistency. The user is presented with an notification asking what to do with the situation: continue cancel the action.

- When a model is saved, the default GMF implementation serializes this to XMI. The default behaviour is to identify element using the relative location in a file. Because the tracing-toolkit will be embedded within an Eclipse IDE, using multiple projects at once, also multiple models (in different projects) can be opened simultaneously. To achieve consistency checking, the element in each model need to be uniquely identifies, also over multiple models in a project. To achieve this, the Tracing Toolkit offers a custom identification manager that assigns (project wide) unique identifiers to each element in the model. This manager is called upon the save action of a editor, reassigning the unique identifiers of the model, ensuring uniqueness of each element within a projects. How this is achieved is discussed later-on

Figure 27 displays a screenshot of the requirements editor. We see here that 3 requirements are displayed, in which they are all satisfied (displayed green, and has a 'Satisfied' text). In the screenshot a satisfied element is being deleted, and the question is asked whether to continue, because the element is still satisfied. By which element the requirement is satisfied is seen in the properties view (bottom).

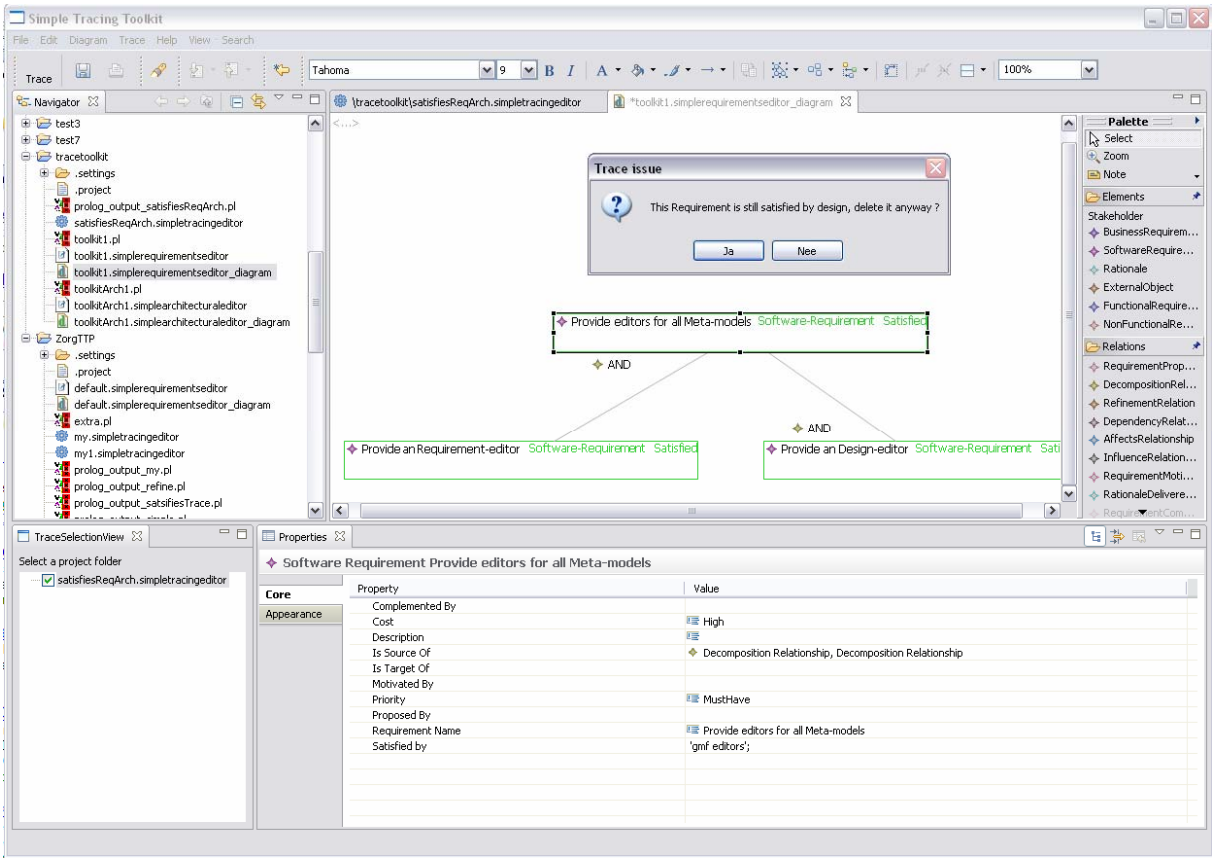


Figure 27: Tracing Toolkit: requirement editor

For more screenshots, showing the requirement and (architectural) design editor, see [Appendix E. Screenshot of the Tracing Toolkit]. These are screenshots of both editors opening the ZorgTTP and TMS models, used to validate the requirements metamodel and architectural metamodel respectively.

Create (trace) relations

The Tracing Toolkit offers a trace-editor to create the traces between the model elements. To achieve this, the editor features a trace-matrix representing the model elements on the axis of the matrix. A trace is created by double-clicking on a cell in the matrix, and selecting the appropriate trace relation to be created. This creates a trace relation between the model on the x-axis and the y-axis representing the connected models. In order to create a new trace model, the user can access a wizard to select the models, being represented on the both axis of the matrix, as seen in Figure 28. This created the empty trace matrix, in order to relate the two selected models.

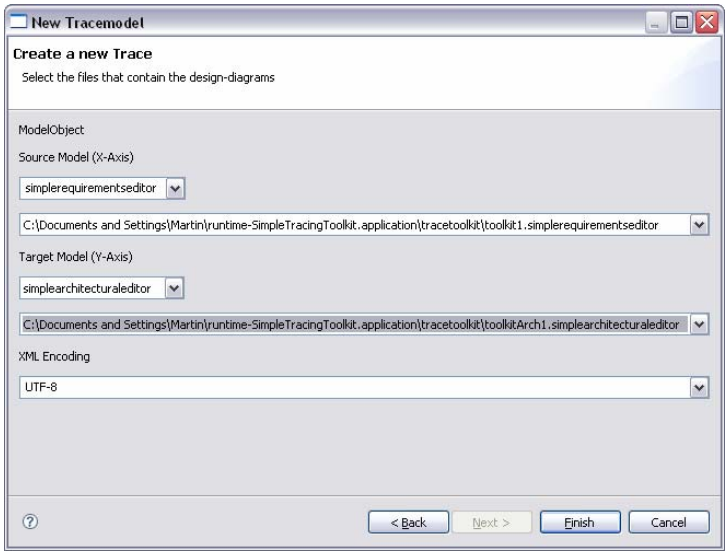


Figure 28: Tracing Toolkit: A Wizard for creating a trace model

Figure 29, displays a screenshot of the trace editor. In the bottom a properties view is given, to display the current trace relation that is created in the cells of the trace matrix. This editor saves its information using the EMF framework.

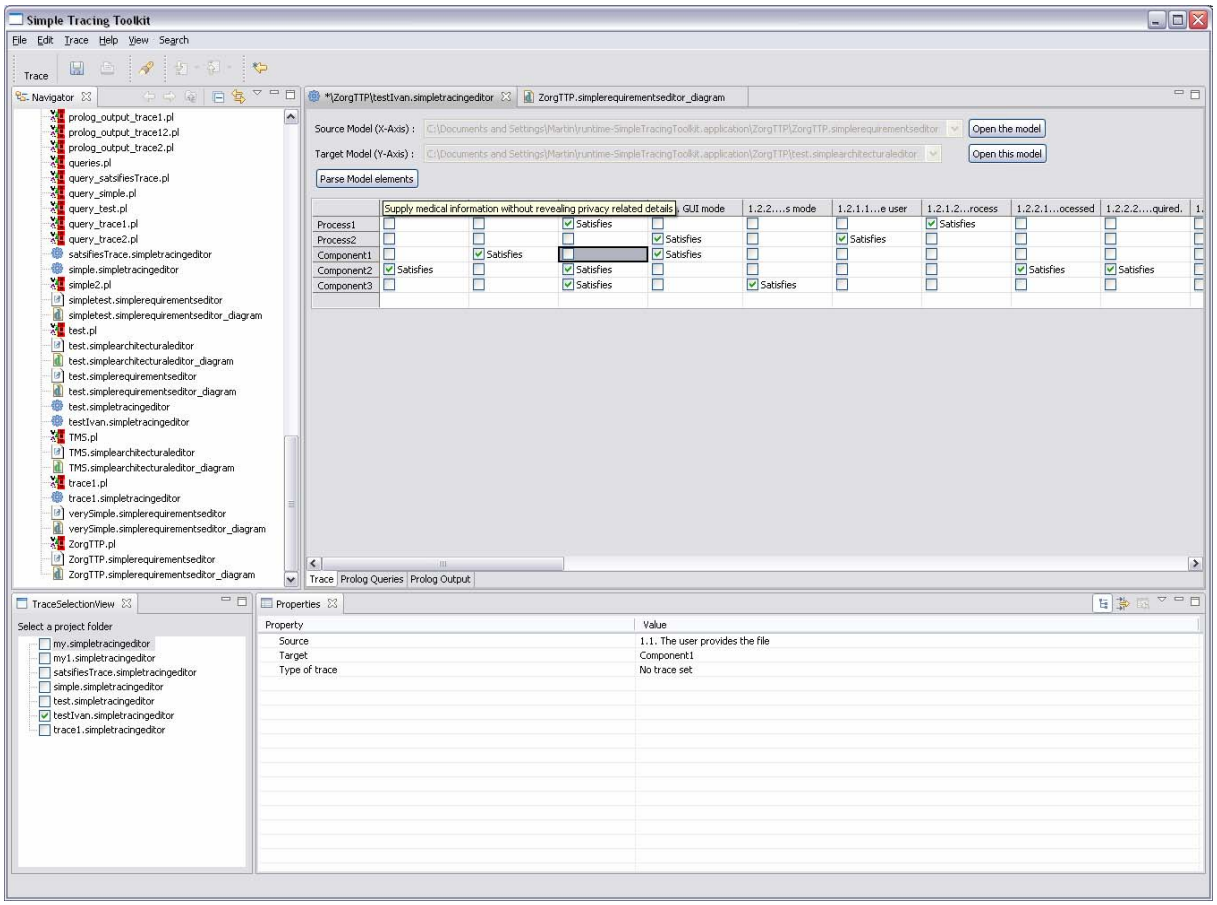


Figure 29: Tracing Toolkit, Trace editor

6.4.3 Storing model information

As said before, all the model information is saved in a XMI format, using the EMF framework. In order to uniquely identification each model element, the Tracing Toolkit extends this default behaviour with the identification manager. To achieve this, the Tracing Toolkit implements a custom approach to identify each EMF element. It uses the Universal Unique Identifier (UUID) attribute that can be assigned to each Ecore class. This UUID attribute can be automatically assigned by EMF, but is then only unique within a model, and not within (a) project(s). To achieve this global uniqueness, the Tracing Toolkit offers a UUID manager that assigns these UUID's uniquely over projects upon saving a model. Each element is identified as follows:

model_type_ projectName_modelName_ElementNumber

The model type is either a requirement design or trace model. The project name is the name of the projectfolder in the Eclipse navigator. The model name is the file name of the model itself. And the element number is a unique number of each element within a model. Combining these features, each element has a unique identifier. An example model serialisation can be found in [Appendix F. Example Model Serialisation]. Each type of model is saved in a separate file containing the XMI information of that model. In this way separate editors can be created for each type of model. If later-on a different editor needs to be created for one of the model types, this can be done. It would require an editor to use the EMF structure to access the information, and represent it to the user in a different form. Using this separation of files (and concern), the Tracing Toolkit is highly extendible for the future.

Next to the serialisation to XMI, the Tracing Toolkit also performs a serialisation of each model to Prolog facts. This is performed to enable reasoning about the consistency of models, using Prolog (described next). In order to do this, the Tracing Toolkit features a custom Prolog serialiser that generated a Prolog fact for each element, attribute and relation found in the models. The serialiser is generic for use in every EMF model. In this way it is used by all the editors in the Tracing Toolkit. The serialisation takes place as follows:

- Each element is serialised as: *class_className(UUID)*
  The className of the element represent the class of which the element is an instance. The UUID is the uniquely assigned identifier of that element. E.g a requirement in a model:
  class_softwarerequirement('r_tracetoolkit_toolkit1_softwarerequirement_0').

- Each attribute of an element is serialised as: *attr_attributeName_(UUID_element, attributeValue)*.
  The attributeName represents the type kind attribute. The UUID_element represents the identifier of the element to which it is associated. The attributeValue represents the value of the attribute. E.g a name attribute of a software-requirement:
  attr_requirementname('r_tracetoolkit_toolkit1_softwarerequirement_2','provide an design-editor').

- Each relation is serialised as: *ref_relationName_(UUID_SourceElement,UUID_TargetElement)*
  The relationName represnts the kind of relation. The UUID_SourceElement represents the source of the relationship. The UUID_TargetElement represents the target of the relationship.
  E.g a software requirement contained in a requirement model:
  ref_containsrequirement('r_tracetoolkit_toolkit1_requirementmodel_0','r_tracetoolkit_toolkit1_softwarerequirement_2').

An example of the Prolog serialisation of the XMI model in [Appendix F. Example Model Serialisation], is given in [Appendix G. Example Prolog Serialisation]. The serialisation supplies us with a way to create a Prolog knowledge base containing all the information of each model. This knowledge base can be queries upon using Prolog. A simple query to get the description of a model element, by supplying its name can be formed as follows:

```
description_of(X,Y) :-
      attr_requirementname(Z,X),
      attr_description(Z,Y).
```

When supplying the name of a requirement in variable X, it will return its description in variable Y. This is a very simple query, but we will see the use of complex queries, using Prolog when discussing the consistency checking of the Tracing Toolkit.

In order to use Prolog as a way to reasoning about models, the Tracing Toolkit uses a Prolog API to communicate with the Prolog SWI engine. To achieve this coupling, the tool has a separate prolog interpreter that handles the translation of the Prolog input and output to Java and visa versa.

6.4.4 Consistency checking

The biggest difference between current existing tooling and the Tracing Toolkit lies in the area of consistency checking. The Tracing Toolkit can, based on the relations in the trace metamodel, check whether two models are consistent with respect to other models.

The Tracing Toolkit performs consistency checking by translating the formally defined trace relations into Prolog queries. Prolog is a declarative, logical programming language, suitable for performing reasoning about facts, present in a knowledge base. This knowledge base is generated by the tracing-toolkit, containing all the information present in the models. It uses this knowledge base to query information about these facts. Prolog queries can be highly sophisticated, and are executed efficiently.

The Tracing Toolkit incorporates a external Prolog engine to check the consistency of the models, using queries based upon our formal definitions. Queries are formulated to check the satisfies and refines relations between models. These queries are executed upon the knowledge base, when a certain model is opened by the editor. It checks for every element in that opened model, whether it is consistent. For a requirement this means checking whether an element is satisfied by an architectural element. And for an architectural element, this means checking whether it has an originating requirement. The connections between requirement and (architectural) models are made by defining the traces in the trace editor (which are also serialised to Prolog).

The Tracing Toolkit supports transitive refinement of architectural models. This means that it can check whether a requirement is satisfied in an architectural model, even if this model is a refinement of other architectural models. It does this, by recursively checking all the refinement relations between architectural models. This is only possible because the toolkit incorporates the use of a Prolog based reasoning engine, enabling formulating complex queries upon the trace relation. The Prolog engine performs the actual resolution (solving) of the correct answers based upon the queries.

An example of the query that checks whether a requirement is implemented (architectural) design is found in [Appendix H. Example Prolog query]. This query supports transitive refinement of design models. The query, requirementImplementation, has as input parameters the requirement to be checked, R, and the architectural model, AM, in which the requirement is possibly implemented. It will return a list, L, containing all the components satisfying this relation. If AM is used as an output parameter, it will return all design models (connected by refinement) which have elements satisfying this requirement.

The use of Prolog offers the power to extend the tooling in the future. When e.g. tracing is extended to implementation level or other kinds of traces are defined, checking these traces only needs to be implemented in Prolog queries. The knowledge base and mechanisms to generate these will remain stable. This is a great advantage over existing tooling which would often require great (internal) changes to incorporate the extensions. The modular architecture, combined with the reasoning by Prolog offers us the possibility to be extensible and offer highly sophisticated reasoning upon traces.

6.4.5 Achieving consistent models

It is outside the scope of this thesis to define process support to maintain the traces, and as a result maintain consistency. We can however explain the steps that it takes to use the tool in maintaining consistency between requirements and architecture. We will describe the steps to perform:

- Create a requirements model
- Create (architectural) design models
- Create a trace model

- Define the satisfies trace trace relations between the requirement model and architectural models.
- If needed, refine the architectural models into other architectural models by each time defining a refines relation between the two.

The orders in which these steps are performed are not really important. The models do however need to be present before traces can be created between them. Once the traces are created, the user can select which traces to actually use in the consistency checking. This is done by selecting the traces in the lower left view of the tool, as seen in Figure 29. When a trace is not selected, it will not be use in the consistency checking. The result of the consistency checking is presented in the actual models themselves by indicating per element whether it is consistent (Also seen in Figure 29).

6.5 Strengths and weaknesses of the Tracing Toolkit

Having discussed the Tracing Toolkit support that has been implemented, we can now discuss the strengths and weaknesses of the tools. We will do so in consecutive order.

6.5.1 Strengths

The strong point of the Tracing Toolkit, discussed in this chapter, will be based on our own initiatives to develop the tools. As a result, this section will be more or less a rationale for designing the tools. We will explain our motivation behind designing the Tracing Toolkit in this way by describing the advantages that it offers. In the next chapter, we will offer a comparison of the Tracing Toolkit, with another tool recently developed to support traceability. In this comparison we will provide a framework for comparing the features of the Tracing Toolkit, which will result in positive and negative aspects of both tools. This comparison will be more objective than the strengths mentioned in this section, basically describing the ideology of the design of the Tracing Toolkit.

Based on models, inspired by MDA & The use of generated code

The Tracing Toolkit is highly driven by models. As discussed earlier, the editors used in our toolkit are based on the metamodels designed for the requirement, (architectural) design and trace models. These editors enable the user to edit instances of these metamodels in a graphical way. These models become the basis for establishing trace-links between artefacts in these models.

A side-result of the model-driven approach is that it enables the (partly) generation of code. By working with metamodels, we generated similar editors for both the requirements and architectural models. Next to a uniform look and feel, this also eases the integration of these editors into one toolkit. Because the trace editor is also created using the (EMF) framework, we can treat all models created in our tool uniformly. This offers a great advantage in saving the model information to disk, but also when transformation needs to be made to other programming paradigm like Prolog. It also eases the development of a trace editor to create traces between the model elements. Creating traces between originally heterogeneous modelling artefacts can now be achieved by creating links between (homogenous) EMF artefacts. Because the artefacts of all the models are derived from the EMF Ecore model, we have a general ‘interface’ to treat those elements in a uniform way. This allows us to create traces, independent of the related models.

This uniformity in model structure also eases the development of new editors for the models, because they can all use the same skeleton code (created by EMF), leaving the rest of the application unaffected. A new editor for either of the models can easily be integrated.

Use of multiple programming paradigms

Combining two different programming paradigms like Java and Prolog in developing our toolkit enables us to take advantages of the unique strengths of both. The Java language offers us a highly supported and state-of-the art programming language to develop an fully object oriented application including a GUI. By furthermore using the Eclipse RCP platform, offering a rich set of tools to build our application on, we end up with a rich programming environment for our application. An advantage of the Eclipse platform is the possibility to use the EMF framework, including its ability to generate graphical editors for modelling our input models. Java and the Eclipse RCP are a powerful application

framework to input our models into a common data structure and perform operations on it. Once the models and the trace information is put into the system, we need to check the consistency of the models. To achieve this, we have to reason on possible implication that changes in a models might have (on other models). We have chosen another programming paradigm, better suited for performing queries on a given knowledge bases, for this purpose. The programming language Prolog enables us with the reasoning capabilities of a logical programming language. Prolog offers the ability to ask complex queries about the so called facts and relations amongst these facts, stated as a Prolog program.

Prolog is a declarative programming language in which the program is expressed by declaring relations amongst terms. These terms represent the knowledge that the system has about the world. A Prolog program is executed by stating queries on these facts and relations. Prolog will attempt to find a resolution to the query and will return the result. It is very much used in database driven applications, in which large collections of data need to be queried in a efficient way.

Prolog has however also some downsides. It is for example not very much suited for graphical representations of data, or for building complex object oriented data structures. The only input and output capabilities it offers is through impure-predicates. Impure-predicates are predicates that create so called side-effects. These side effects are the functionalities often seen in imperative programming languages such as printing to a screen or writing and reading a file to disk. These capabilities are very limited, as the Prolog language is not designed to be a fully fledged graphical I/O system.

In order to make use of the powerful querying capabilities that Prolog offers, and be able to use the capabilities of Java and Eclipse, we have chosen to ‘embed’ the Prolog language in our Java built toolkit. We achieved this by making a simple model transformation to facts and relations amongst these facts, based upon information present in our models. In this way, Prolog is able to understand the information that the user has put into the models. Once these facts and relations are available, we call the Prolog Engine from within the Java environment, and pose queries on them in order to check the consistency of the models.

By using a wrapper to access the Prolog engine, we are using the full capabilities of the Prolog reasoning from inside our Java application. The output of the Prolog engine is accessed by the Java program, to be able to represent the results of Prolog in a convenient manner.

In our specific case, we use the Prolog engine to ask queries about possible implications that altering a model element can have. Once we have a result of the impacted model elements, we now represent this information in a graphical way, using the editors provided by the Java environment (as described in previous section).

Improved consistency checking

As already shortly mentioned in the previous section describing the consistency checking in the Tracing Toolkit, the biggest advantage of our tooling over existing tooling lies in the area of consistency checking.

The Tracing Toolkit offers formally defined semantics of the trace relations, which is often lacking in other tools. We have formally defined these relations in Chapter 4. We defined what it means for a model to be consistent with respect to other models. We have also defined the refinement relation between models. Defining these relations is possible because we have a separate trace model, offering typed trace relations, differentiating between separate kinds of traces. The Tracing Toolkit uses these formalised relations in order to check the consistency of models. This is achieved by incorporating these formally defined relations into Prolog queries which can check the correctness of these relations. In other existing tooling, these relations are often implicit, and also not formally defined.

Extendibility of the tooling

The Tracing Toolkit is highly extendible in various ways. First, because the editors for the models, rely highly on generated code, they can easily be replaced or extended by other editors. This can be a big advantage when other editors need to be created. When the changed editors will remain using the

EMF code, they can be incorporated into the existing Tracing Toolkit without adjusting other components. This is mainly because of the component based nature of the tool (using plug-ins) and the use of EMF as a basis for every editor.

Another area in which the tool can be extended is in the metamodel. Because the editors are mainly generated, these can also be regenerated when the metamodels are changed. Depending on the changes in the metamodels, this would require little changed to the tracing mechanism, because this is highly based upon the generic EMF, Ecore objects.

Next to this, also the capabilities of the consistency checking can be easily extended. The trace metamodel support the adding of additional trace relation types to use in trace analysis. The trace editor can easily be adjusted to add additional types of traces. Furthermore it would require a formal definition of the new trace relation to be established. Once this is known, a Prolog query to check the correctness of this relation can be made, and easily integrated in the tool. The query would then only have to be called from within the tool, to complete the new consistency checking method.

Although we have not experimented with within-model relations as traces, this should also prove possible. The relations are already defined in both the requirements and (architectural) metamodel. They only need to be treated as traces. The trace metamodel already supports this. It would however require changes to the trace editor, which needs to recognise the existing relations in a model as traces. Additional research in how this can best be achieved is needed.

6.5.2 Weaknesses

Next to the abovementioned positive aspects of the Tracing Toolkit, we also have to mention some weaknesses of designing the tool support in this way. In this section we will discuss these inherit weak points that we introduce with the way of working that we have followed in building the tools.

Scalability

One of the problems that arise from the graphical representation of traces in a matrix is a scalability problem. This problem arises when the number of elements in the traced models becomes larger. As a result, the number of elements on the axes of the matrix will also become larger, and might cause the user of the matrix to loose track of the situation. This problem is not a new problem, and is also seen in a number of commercial tools that also use matrixes to represent trace relations between different models.

A possible solution to this problem can be to (temporarily) reduce the number of elements that are represented on the axis of the matrix, to suit a specific purpose. This can be achieved in a number of ways. One possible option could be the use of filters to hide specific elements which are not relevant at that time. Another possibility is to use the decomposition mechanism built into the requirements and architectural models to hide certain details. The level of decomposition can then be used as a selection criterion, to select which elements to show on the axis of the trace matrix.

Another area in which scalability plays a role is the editing of requirements in a graphical way. Although with a limited number of requirements, graphical editing of requirements can be very insightful, a problem may arise when the number of requirements will exceed even 100 requirements. When these large numbers of requirements are modelled in this way, the models tend to get very large, in which the overview is easily lost. We will propose a possible future solution to this problem in the next section.

Generation is still in its early phases

Another problem we experiences while building the tool support is the immaturity of the code generation frameworks. Especially the Graphical Modelling Framework is still very much in it's early development phases, and is not yet very mature. This is noticeable in the quality of the tooling, which sometimes has unexpected behaviour. This makes working with the GMF framework labour intensive and troublesome.

6.6 Future extensions and improvements

During the development of the tool support we could already envision a number of future extensions to the tool support, which are currently not implemented because of time constraints. These future extensions can however improve the functionality and usability of the tool.

Other editors

The current editors available to edit the models are graphical ones. This is mainly because this eased the development, through the generation of editors. These editors are in some circumstances however not always ideal to use. As we mentioned earlier, the graphical requirement editor has for example some scalability issues.

A future extension can therefore be to use different editors for these models. As mentioned before, the modular nature of the EMF framework incorporated in our tools, eases the implementation of a new editors based on this framework. To overcome the scalability problems in the requirement modelling, a tree based or even textual editor would be a logical solution. Further research is however needed before statements about this can be made. Future editors can still be based on the requirement metamodel which we propose in this thesis. When these new editors remain based on the EMF framework, and still use the XMI-structure to store their models, they can easily be integrated in our toolkit.

More trace relation types

Currently, we have only implemented two types of (interlevel) trace relations in our tooling: Satisfies and Refines. It should however be possible to define other trace relations types between model elements. When for example lower level design models need to be incorporated, relations between the elements in these models need to be implemented. Another important trace relation that needs further research is the evolve relation between model elements. In this thesis we have specifically excluded this relation because it is a highly unknown area of research and falls outside the scope of this thesis. Future improvements of the tooling could however include this relation.

Another future extension to the trace relations that can be expressed are the within-model relations. Currently we have not implemented these relations as possible traces in our tool. It should however be possible to use these existing relations that are present in the requirement and architectural models as trace candidates in the trace matrix. One simple implementation could just iterate through all the possible relations in these models, and present these as suggestions in the trace matrix when two of the same instances of a model are selected in creating the trace matrix. We will give a number of possible extensions using other relations as a trace:

- Using within-level relations: Between level relations can be useful to analyse certain structures in a model. The decomposition relation, often seen in requirement documents can be used as a trace relation. This can aid a developer who gets the task to implement a certain high level requirement to track down the lowest level of requirements to start implementing.
- Combining within-level and interlevel relations: The relations between the constructions made in the within-level relations and the interlevel relations can also be checked. An example of this kind would be to check whether the decomposition construction is consistent with the design level. If for example a requirement is decomposed in a number of sub requirements using the XOR construction then only one of the sub requirements is allowed to be implemented. This can be checked using the within-level decomposition related to the satisfies inter level relation. Another extension of this kind could be to check whether no two excluding requirements are both implemented in one design.
- Using intralevel relations: An intralevel relation could be used to check consistency between different models describing similar information. Because the intralevel relations are defined amongst model in the same development phase, they will often relate similar models. An example of would be to check whether in two models connected by refinement, every element in the source model is actually refined in the other model.

Document generation

A totally different use of our proposed metamodels and editors for creating instances of these metamodels, is the generation of documentation according to predefined standards. This enables development companies to adhere to different requirement standards, based on the information available in the requirement models. Basically, a model-to-text transformation needs to be made from every element and attribute in the provided metamodels to a certain template.

Generating these documents removes the manual labour intensive writing of requirements documents, once the information is available in the system. In the current practice it often occurs that official requirement standards are used only to satisfy the customer or certain legislations. Next to these documents, often a separate document/model is maintained to be used by the developers. This means that a lot of work duplication is carried out. When these documents can be generated according to a certain standard, different representations of the same requirement information can be created, each satisfying a specific stakeholder. This removes the work duplication and ensures that each stakeholder can use its own requirement document representation as he/she is used to.

Improving scalability

The editors available in the current tooling suffer from a certain scalability problem when the number of element in the models becomes larger. Solutions to these scalability problems are however dependent on the kind of model being edited. We will shortly discuss some possible extensions that can improve the scalability problem per model:

- Requirements model: Requirement documents tend to be long and very detailed models describing the desired functionality in every detail. This is why scalability is a very important aspect in designing editor for requirement documents. Nowadays, requirement documents are often edited in text editors who often represent the requirement model as a big listing of texts, often structured by decomposition (From very general descriptions downwards to detailed small sub requirements). In order to be as close to this representation as possible, creating an editor that treats requirement elements as lists could be a solution. One way would be to represent them as a tree based structure, dictated by decomposition. The danger is that this puts the other relations formed between requirement elements to the background. Another option would be to make a text based editor which can be interpreted by software to create the mapping to the offered requirements metamodel. This has the downside that it is not very user friendly. The user has to know the ways to tag certain text elements with the correct requirement model elements. An XMI based editor would in this case be a good alternative, because here the assigning of requirement model information to the text is natural. User support for building the correct XMI-metamodel should then be available to assist the user in selecting the right tags. Automated support for checking the correctness of XMI according to a certain metamodel is widely available.
- Trace model: As described earlier, a useful extension to the current tooling would be the implementation of filters on the axis of the matrix. These filters can be custom made, or can even use the existing decomposition mechanism to reduce the number of elements on the axis. Another possibility to reduce the amount of elements on the axis of the matrix is to use a treelike system of expanding detail when needed. Only when needed, these details are made visible to the user of the system.

To lower level design elements

Ultimately, the tools should be expended to support every phase in the development process, ranging from he initial business requirements to the actual implementation in source code. Traceability should be available throughout this process. We have in this thesis only focussed on the top two phases. It should however be possible to extend this to lower level design elements. Ultimately, it should be possible to relate business requirements to the source code blocks which really implement this functionality. In order to support this, also the intermediate levels of detail should be supported. A first step would be to extend the traceability to detailed design (e.g. a detailed class diagrams). When this functionality is in place, the step to actual source code is fairly small.

A big advantage of implementing these final steps would be that it offers the possibility to check if all requirements are ultimately implemented. Visa versa it could be checked if no unneeded code is implemented (e.g. is implemented without any need for it).

Implementing the lower level design levels into the tooling would open up an even greater world of possibilities of using trace information. Questions like which requirements lead to the biggest source code blocks suddenly can be answered. This kind of information can be used to improve future development projects. This information that in the past always has been hidden in project documentation can enable managers to take better decisions in future development trajectories.

Defining a process for using the tooling

In order to make the most use out of tooling to support consistency checking, process support should also be available. Further research is needed in order to define a process that defines how consistency checking can best be performed (using tooling). This could provide a description of how to optimally use the tooling. It is however outside the scope of this thesis to investigate this.

6.7 Summary

In this chapter, our tooling for performing consistency checking between requirements and (architectural) design is introduced. We first introduced the Eclipse RCP, which forms the basis for our tool; the Tracing Toolkit. We introduced the model driven frameworks of the Eclipse platform used to create our tooling; EMF and GMF. These components have been used to (partly) generate source code for our tool. When describing the architecture of the Tracing Toolkit, we describe how all the components of the tool work together (generated and custom ones). We have built the tool using a combination of existing components and custom developed components. In order to describe how Model Driven Development is related to our tooling, we describe the use of metamodels in our tool, describing how this relates to OMG MOF.

A complete section is dedicated to describing the functionality of the tool. We have done this by describing the requirements of the tool, and explaining per requirement, how these are satisfied in the Tracing Toolkit. We described how we achieved the integration of the tool into the Eclipse environment, using the Eclipse RCP.

We described the behaviour of both the requirements and architectural editors in representing information concerning the consistency of the model. These editors save their information, using standard XMI serialisation, extended with ways to uniquely identify each model element. This is achieved by implementing a custom identification manager. How this is an achieved has been described. Next to the editors, we also discuss the custom built trace editor and how this is used to create traces between models.

We described how the editors, serialise their information to a Prolog knowledge base. Using this Prolog knowledge base the tooling actually performs the consistency checking. It uses queries to check the formalised consistency of a model, upon the generated knowledge base. It is also quickly described how the tool can be used in an actual development process. It requires however further research to define a complete process to support the consistency checking using our tools.

The chapter includes a description of strong and weak points of the tool. Strong point include: The highly model-driven basis of the tool; The use of multiple programming paradigms taking advantage of the strengths of each of them; The improved consistency checking using the formally defined and typed trace relations; The possibility to extend the tooling in the direction of editors and consistency checking. The weak points include: Problems with the scalability of the editors and problems with the use of an immature generation frameworks, like GMF.

The chapter concludes with a description of future work in the area of tooling. These include: Further research in better suited editors; Investigation into the extension of the types of traces; Inclusion of a document generator; Improving scalability; The extension of consistency checking towards the lowest level of design; And finally, the definition of process support for using the tool.

7. A comparison with existing tooling

After having discussed our approach to improve the consistency between requirement and lower level design artefacts, it is interesting to compare our approach with the approach taken by Chess in their Forest project. The Forest project also involved the building of tool support to improve amongst other things, the traceability between requirements and implementation. We will discuss the Forest approach in more detail in this chapter. In this chapter we will mainly focus on making a comparison between the tooling offered by the Forest project, and our approach. In order to do this, we will first introduce a comparison framework on the basis of which we can compare the two approaches in Section 7.1. After discussing the comparison framework, we will shortly discuss the tooling of the Forest project, in order to get an idea on the way of working in their project (Section 7.2). We will make a comparison of both tools on the basis of these predefined criteria in Section 7.3. Section 7.4 will finally present the conclusion of the comparison between our tooling and the tooling of the Forest project.

7.1 A comparison framework on traceability tools

Devising a comparison framework for an immature domain such as trace tooling is a challenging task. One of the main reasons for this is (again) the lack of consensus on what trace tools should do. This problem is directly related to the lack of consensus on definitions in traceability. Again the main cause of these problems lies in the fact that traceability is highly related to the goals that stakeholder have with it. We have however a concrete goal with traceability, being consistency checking. We will therefore focus on this concept when comparing both tools.

In order to come up with a comparison framework to compare the Forest tooling with our own tooling we used a number of resources. The first resource is the original requirements defined at the start of the Forest project [Appendix B. Requirements Forest project]. These requirements can be used to compare how both tools relate to these targets. Also our own project has some requirements associated with it [Appendix C. Requirements Tracing Toolkit]. We will take both these requirements as one source, and combine them when they overlap. Next to these requirements, we also addressed a number of general criteria for tooling; usability, extendibility and scalability.

Before we can use these criteria, we first define the meaning of them. The definitions of usability in the ISO 9126 standard referrers to usability as: understandability (effort to recognize the concept and applicability), learnability (effort to learn its application) and operability (effort for operation and control). We will focus on the operability, defined in the ISO 9126 as "*Attributes of software that bear on the users' effort for operation and operation control*" [SSE'08].
We will define extendibility as "*The ease with which a system or component can be modified to increase its storage or functional capacity*" [IEEE-610'90]. We will mainly focus on the ease with which the system can be modified to increase its functionality. Increasing the storage capabilities will be addressed in a separate criteria covering storage of the tools.
Scalability finally is defined as "*The ease with which a system or component can be modified to fit the problem area.*" [SEI'08]. We fill focus on the ease with which the system can cope with a growing problem area (in requirements, architecture and traces).

Furthermore we also addressed criteria which are specific for tooling in the traceability domain. Because this is a rather immature domain, we cannot specify standard definitions for this. We will here focus on the creation, representation and specification level of traces.

The full comparison criteria, grouped per source of criteria, can be found in appendix D [Appendix D. Comparison criteria for comparing Forest Tool with the Tracing Toolkit]. As clearly visible from these comparison criteria, there is some overlap amongst the grouping, especially in the requirements. In order to arrive at a non-overlapping comparison framework, we will first perform a general categorisation, and remove any redundant elements, and combine them in a common denominator. The resulting comparison framework is presented next.

- Storage
    - Storage location: *where is the information of the tooling stored.*
    - Kind of information stored: *which kind of format is used.*
    - Openness of the storage format: *does the tool use a open or proprietary format.*
    - Extendibility of the storage format: *can the storage format be easily extended.*
- Trace checking
    - Creation of traces: *how are traces created in the tool.*
    - Kind of analysis to perform: *what analysis methods are available*
    - Kind of (automated) traceability check: *how does the tool use the trace checking*
    - Extendibility of analysis: *can the analysis of traces be extended*
    - How to perform the analysis : *Automated / Manual*
- Usability (following the definition)
    - Creation of requirements and design
    - Creation of traces
    - Easy of use through non-mandatory information. (Use as see fit)
    - Defining of (sub) systems
    - Integration with the Development environment
    - Reuse of artefacts
- Extendibility (following the definition)
    - Support for different traces to be created
    - Support for different editors
    - Generation of documentation
    - Level of detail in modelling
    - Freedom in Relations
- Scalability (following the definition)
    - Requirements editing
    - Design editing – subsystems in Forest
    - Representation / Creation of traces: trace matrix is generated by Forest.

7.2 The Forest Eclipse plug-in

The Forest project is a project initiated by Chess, a software development company at Haarlem as a new method of creating and maintaining documentation in a development project. As a part of this project, a well-defined repository and a set of tools to store information into this repository in a structured and traceably manner, is required [Kraus'07]. The Forest Eclipse plug-in is the actual solution to the required tools for the Forest project, consisting of a set of smaller tools which are aimed at creation and maintenance of system documentation. The tool assists in entering and storing this information and relations between them in a structured manner. The tool enables the users to perform certain checks on this structured information, to verify that the information is consistent and complete [Kraus'07]. Furthermore, the tool also allows for the generation of personalised documentation based on the information in the tooling. This documentation can be adjusted to address the needs of the specific stakeholder of the documentation, providing a flexible document generator based upon templates. [Kraus'07]. Traceability checks, performed upon a system can be used to include into the documentation to for example indicate the satisfaction of the requirements.

7.2.1 The Forest repository elements

The repository of the Forest is the component responsible for storing design information on the basis of which the document generation and trace checking can be performed. This repository stores both the requirements elements and design elements into a single repository. In fact, all development related artefacts are stored inside the Forest repository. In this subsection we will explain the main elements used in this repository and explain how it is used.

Requirement

A requirement element contains details of a requirement, such as an identifier, a title and a textual description. The requirement element is used to represent every kind of requirement ranging from business requirements, functional requirements to lowest level software requirements.

System

A system element is a representation of a design artefact. The system element is modelled as a composite element. A system furthermore contains a number of requirements and a number of atom elements. A system can be keyed with the keyword 'abstract', meaning that the system does not need to be implemented in some kind of Atom element. An abstract system can be extended by another system containing at least the requirements and subsystems of the abstract system. The extending system must, not being abstract, implement the requirements connected to it. A system can be specified in a separate file through the use of a location attribute. This enables a designer to keep the detailed sub design of a complex system separate from the design of the main system.

Atom

The atom element represents the lowest level of a system. It represents an atomic piece of implementation [Kraus'07]. An atom is ultimately responsible for implementing a requirement. An atom can furthermore be linked to an actual source code artefact which implements the atom. An atom can also be used to represent a piece of design which is not elaborated on further. This atom is then responsible for implementing the requirements. When later-on a designer of developer wants to implement the atom into a system, the atom can always be replaced with a subsystem, containing more details.

Delegation

The delegation element is a mechanism to connect requirements to either a system, or an atom. It represents the implementation link of a requirement to its target (sub) system. When a delegation from a requirement to a system or atom is made, this requirement is implemented in the system or atom to which the delegation connects. A (sub) system is never an implementation to a requirement, but more a mechanism to abstract away from the system. A system will always contain atoms which will

ultimately implement the requirements of the system. Delegation can also connect two requirements, meaning that a requirement is replaced with another version of the requirement (e.g. because more details are known).

The delegation mechanism is also the mechanism used to establish 'traces' from a requirement to an implementation artefact (atoms). A delegation chain can be established from a requirement through (sub) systems which inherit these requirements to the actual implementation elements, modelled as atoms. These delegation traces are later on used to check if a requirement is implemented, and whether an implementation element, actually has requirements associated with it. This is depicted in Figure 30, inspired by the work of [Kraus'07]. The arrows represent a trace from a requirement, through a subsystem which inherits the requirement, to an atom. The delegation mechanism is clearly visible and provides the connection of a requirement to an implementation element.

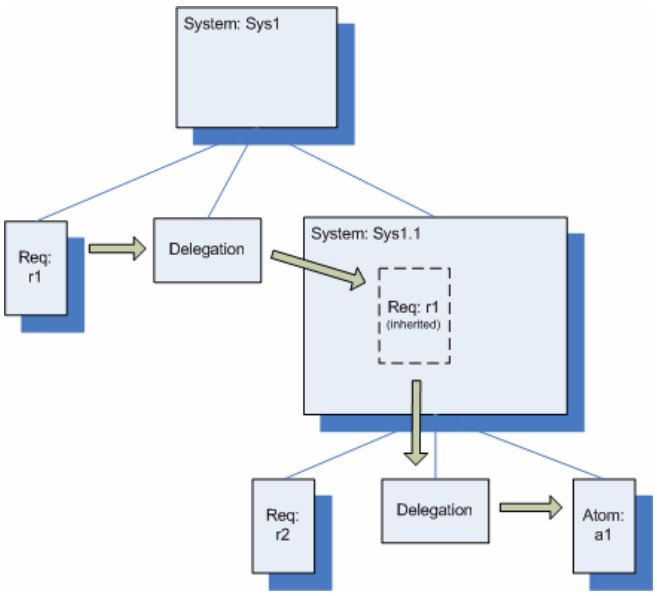


Figure 30: Delegation in the Forest project

Additional elements

Next to these basic elements, the Forest project also includes some additional elements to support the tracing functionality. We will shortly mention them here.

Extends: The extends element is used to indicate that a system implements the requirements of a abstract system. The abstract system itself will not need to implement the requirements connected to it. Any system connected to this abstract system (via the extends element) will need to implement these requirements. The main reason for this mechanism is to offer the possibility to reuse systems, with its connected requirements in multiple situations.

Implements: The implements element is a sub element of the Atom element containing a reference to a requirement that is implemented by the atom.

Source and Target: The source and Target elements are sub elements of the Delegation element, containing references to other elements. The source element typically refers to a requirement, and a target element will typically refer to a atom or a (sub) system.

Title, Description, Motivation and Image: These elements are each represented as separate elements in the Forest model. They are however mainly used as attributes for System, Requirement, Atom and Delegation elements. The titles are quite self-explanatory. These elements are used to clarify the other elements in the model.

A bird-eye view of the Forest repository

The Forest repository is defined in a DTD-based (Document Type Definition) schemata, which is quite difficult to read and interpreter. To offer a model which lies closer to the earlier discussed metamodels modelled in UML, we have devised an informal abstract of the Forest-repository model. We left out some of the relations between elements because these would clutter the diagram with self-explanatory relations. Everywhere where a reference (ref) is mentioned, this usually means that this attribute contains a reference to another XMI element (all classes in our diagram). For example, a Source element is now modelled to contain a reference to a requirement, but this element can technically also refer to an atom. It is left to the implementation of this model to constraint which elements can be contained by a reference. Our abstract model is depicted in Figure 31, and is meant to offer insight into the repository structure, not as a exact model of the actual Forest XMI Schemata.

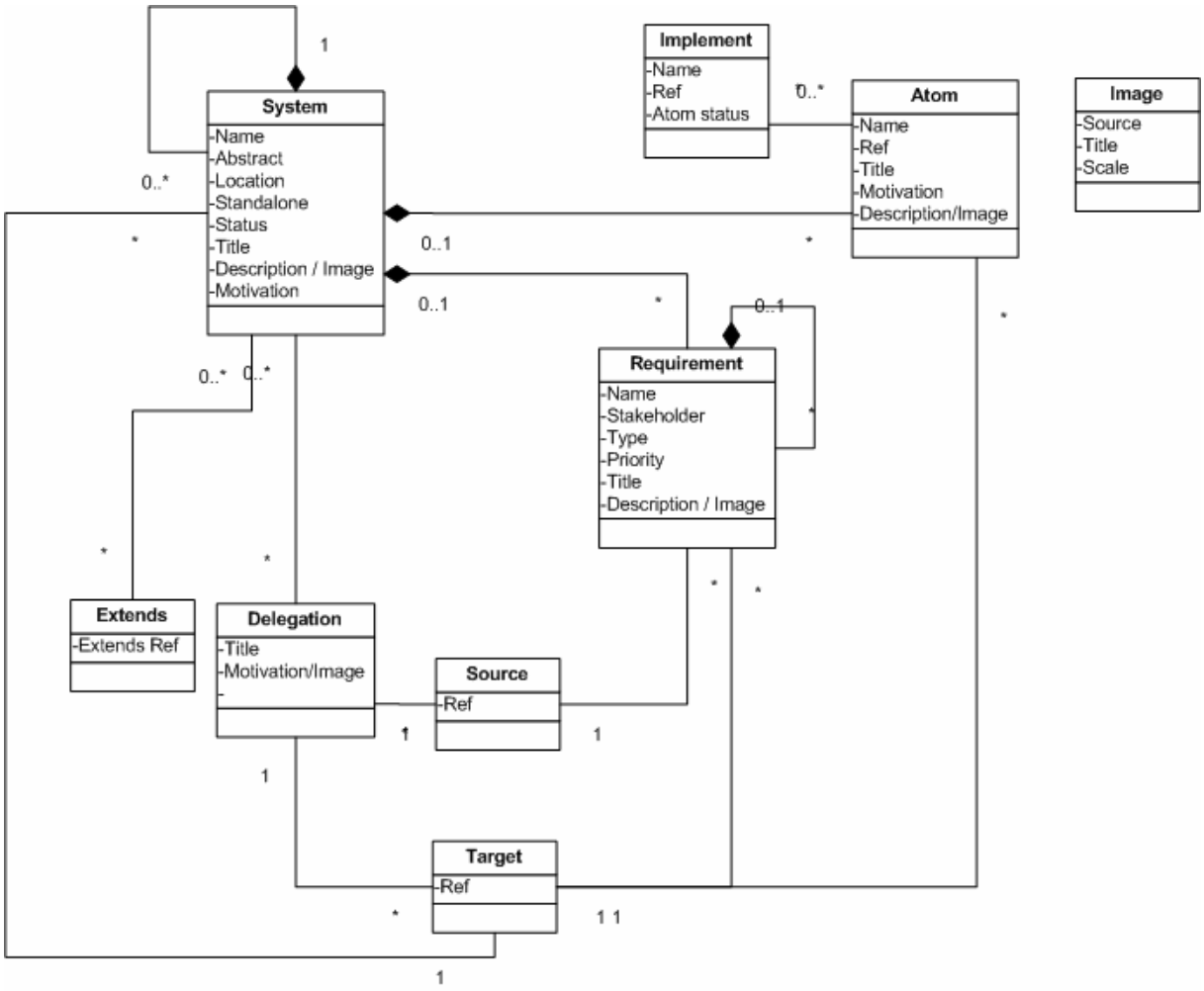


Figure 31: Forest repository model abstract

7.2.2 The Forest tooling

After having discussed the model of the repository in the Forest-project, we will now shortly discuss the tooling of the Forest-project itself. This tooling consists of an Eclipse plug-in which can be installed in the Eclipse IDE. We will discuss the most important components of this plug-in as they are implemented.

Repository

The repository of the Forest-plug-in contains the actual information about the systems being developed. The information that is present in the repository are requirements, artefacts containing

architecture and design descriptions and test case descriptions. The actual information is stored as an XML file in between the source code of the software development project. XML is chosen to comply with the demand of a open and readable storage format.

Editors

Because XML is not a very user friendly format to edit directly, editors are available to edit the information in the repository. The Forest-plug-in offers an editor for this model, based upon a tree view like presentation. Because XML is an open format, it is still possible for any third party to devise its own editor based upon the Forest-model. The tree view like editor of the Forest-plug-in enables the user to insert systems, requirements and atoms into the plug-in. The delegation relation is used to relate the elements. Generally, first a system is modelled that has a number of (high level) requirements associated with it. These requirements are often implemented by subsystems of the composite system. These sub systems will again be decomposed into sub subsystems until atomic implementations can be modelled to satisfy the requirements. The Forest-plug-in momentarily features two editors; one editor to edit the elements individually, and an editor to manage the structure of delegation.

Model Checker

To be able to check certain constraints which need to be true for the repository to valid, the plug-in contains a so called model checker. This checker will for example check if a system artefact always originates from a requirement. The checker considers a repository valid, if all the relations and elements also actually exist in the repository (e.g. no delegation to non-existing systems exist). The plug-in currently supports what they call requirements traceability. Requirements traceability refers to tracing requirements through the architectural and detailed design to implementation artefacts, indicating whether a requirement is implemented. Based upon this requirement traceability, the tool enables the user to perform two types of analysis; coverage analysis and derivation analysis. Coverage analysis, checks whether all the requirements are actually implemented into an implementation artefact. Derivation analysis is the other way around, indicating whether all implementation artefacts also have associated requirements. The tool performs the analysis and gives warnings when a problem is indicated (e.g. some requirements are not implemented). This cover the traceability options of the Forest tool.

Document Generator

To fulfil the requirements of documentation generation, a document generator is included in the Forest-plug-in. It is intended to generate two types of documents; a requirements document and a detailed design document. These two documents are a part of the JSTD-016 standard, commonly used by Chess in their software development projects. A prototype implementation of this generator is included in the current implementation. It is indicated in the documentation that in future versions, the tooling should use documentation templates. This is however mentioned as a future improvement. The current document generator enables the generation of traceability matrixes to show the implementation status of each requirement. This matrix is a direct result of the two possible analysis types which can be performed; the coverage analysis and the derivation analysis.

IDE integration

IDE integration within the Forest plug-in is achieved by building the tool in as a Eclipse Rich Client Platform plug-in, which enables the tool to be used within other Eclipse applications, like the Eclipse Java IDE. The Forest plug-in is as a result just a collection of views and editors (plug-ins) which are activated inside the Eclipse IDE. The tooling consists of two editors; a artefact editor and a structure editor. The former is used to edit the details of individual elements and the later to edit the structure of the composite structure of systems, requirements and atoms. An example screenshot of the Forest tooling is shown in Figure 32.

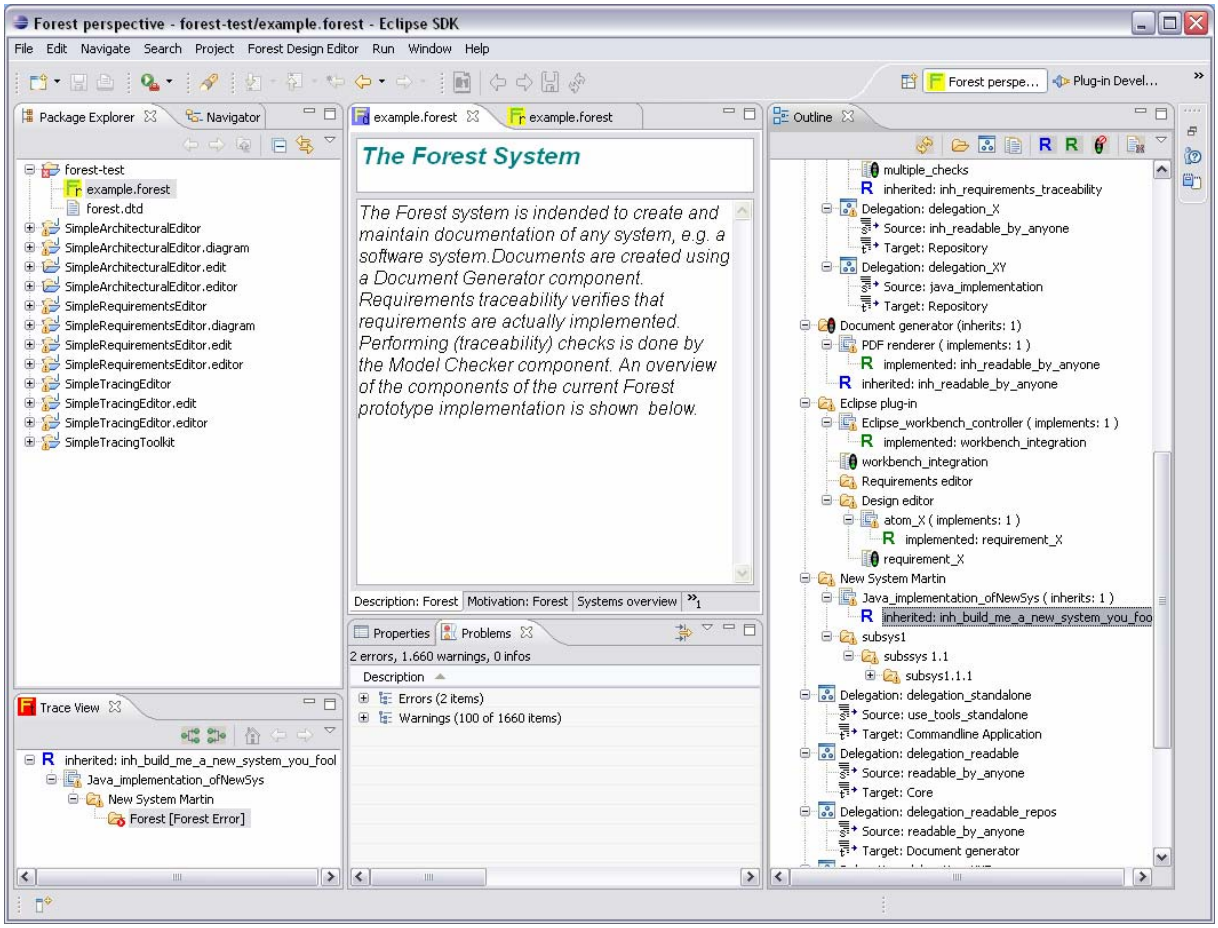


Figure 32: Forest Eclipse-plug-in

7.3 A comparison between the Forest tool and the Tracing Toolkit

In order to provide a structured comparison, we will compare the Forest tool with our Tracing Toolkit according to our earlier mentioned comparison framework. We will however not exactly follow the structure of the comparison framework when discussing the differences and similarities. We have grouped the criteria on a number of common characteristics. We will discuss the comparison according to these grouping.

7.3.1 Storage

Storage location

Both tools store their information in similar ways, both using Eclipse project folders as the basis for file storage. Files are stored within a project, which actually represents a folder/directory in the file systems. Every project is stored in the workspace, being the root directory of all Eclipse projects. The main advantage is that the information of the tools is stored together with any other development files present, such as Java files. When the tool supports tracing to the implementation level, this can be useful, because now the tracing can directly be coupled to the implementation files present in the same Eclipse project. Currently only the Forest tool supports this.

Kind of information stored

The kind of information that is stored differs between the tools. The Forest tool stores all the information that is known to the system in one big XML file. This file actually contains the requirements, design and relations between these two elements. Any supporting elements are also contained in this one file.

The Tracing Toolkit takes a different approach by splitting up the storage of information per metamodel to which they conform. Separate XMI files are created for requirement models, design models and trace models. The trace model is more or less a bridge between two kinds of models, which are being related by the traces. The two related models need not be of different kinds, they can even be the same model, supporting intra level relations (e.g. between two design models). Next to these tree kinds of models, the Tracing Toolkit also generates the Prolog-files used to reason about the traces. These are also stored separately in the project workspace. This enables an experience Prolog user to create custom queries, based upon the facts in the Prolog files.

Openness of the storage format.

Both tools use a similar storage format. The Forest tool uses XML as the storage format, which is being validated on the basis of a DTD file describing the structure of the XML file. When the Forest-model needs to be adjusted, it should be possible to change the DTD, and let the application use the extended XML file.

The Tracing Toolkit uses a combination of XMI and XML to store its information. This is a direct result of using the EMF framework, which uses XMI as a standard to exchange metadata information via XML. XMI is used to store the metamodels which describe the structure of the instantiated models. If the metamodels which the toolkit uses needs to be changed, the XMI model, describing the model definition, also needs to change. All what is left then is to regenerate the editors to handle this changed metamodel.

Extendibility of the storage format.

Both tools, using some kind of metamodel (DTD, XMI) to describe the structure of the stored models, are extendible. The difference is that the Forest tool uses one model for all the information, whereas the Tracing Toolkit uses separate models to capture information of each level of development. Changing for example they way that requirements are captured will have a more isolated impact on the Tracing Toolkit that on the Forest tool. Because the elements in the Tracing Toolkit are also highly decoupled (e.g. a trace model with just references to the other models, and separate editors for each model), changing one of the metamodels will have a very limited impact on the entire tool.

7.3.2 Trace checking

Creation of traces

The creation of traces is quite different in both tools. In the Forest tool, the traces are implicitly created by defining the delegation relation between requirements and systems, until the level of an atom is reached. When a requirement is related to an atom (through a decomposition of (sub) systems), this means that a trace is created. Traces are not specifically created or specified. They are concluded from the absence or presence of a (delegation) relation between a requirement and an atom. There is also only one type of trace to use.

The Tracing Toolkit takes a whole different approach, and requires the user to explicitly create traces, stored in a separate trace model, modelled by a separate trace editor. Traces are always created between a source element and (possibly multiple) target element(s). The metamodel of the trace model provides the possibility to create traces between two model instances, regardless of its type. In this way a requirement models can for instance be traced to design, but also to other a requirement model. When a trace between two model elements is created, the type of trace is also specified. The typing of trace relations offers the possibility to formally define the semantics of the trace relations, as done in Chapter 4. This is actually the basis for the consistency checking. These different types of traces and freedom in the related elements do not limit the analyses that are possible on the basis of these traces
The Tracing Toolkit is designed to also support tracing of between model relations in a requirement or architectural model as a trace. This is however not yet implemented in the actual tooling. Creation of these traces, based upon existing relations should be assisted by automated analysis of these relations. When for instance a decomposition of a requirement to the lowest level of requirement should be treated as a trace, the tool should offer assistance to automatically treat each

decomposition relation as a trace. This would avoid the manual creation of traces between each involved requirement element in the decomposition relation.

Kind of analyses to perform

The analyses that can be performed with both tools are both similar and also different. We will discuss the main differences and similarities per purpose of the trace analysis.

- Coverage analysis: Coverage analysis is a term used by Forest tool to indicate the analysis of implementation of requirements. This kind of analysis basically checks whether the requirements that are present in the system are also implemented. Both tools basically offer this kind of functionality, only in different ways, mainly caused by their different metamodels and differing goals. The Forest tool offers the possibility to check whether requirements are implemented in a atom, representing the implementation level. This is done through the decomposition (or delegation) relations between requirements, (sub) systems and atoms. The goal is to perform some kind of consistency checking, checking whether all requirements are implemented. It is however not formally defined, when a document is considered to consistent. Because there is only one type of relation (delegation), this is also used for tracing purposes. Consistency checking in the Forest project is mainly used to create a requirement document which has indications on the implementation of a requirement.

  The Tracing Toolkit can also perform the same type of analysis, tracing a requirement to an implementation level. One difference is however that the implementation level is currently restricted to design level. Future versions of the toolkit can however provide models toward the actual implementation level of source code. It is just a matter of creating the traces through the levels of development, to the implementation level. Next to this, relations for connecting the elements to these lower level models should be formally defined. Only then can a Prolog query be devised to check the consistency of these relations.

  A big advantage of the Tracing Toolkit is however that it can also check consistency in a more sophisticated manner. The Tracing Toolkit does not only check the direct satisfaction of a requirement in a design model, but can also check satisfaction through transitive refinement. This kind of consistency checking follows the definitions of Section 4.7. This offers the ability to check whether requirements are implemented in each refinement of an architectural design. We will explain this using the example of Figure 21, discussed earlier in Chapter 4. Here we see that a requirements model is initially satisfied by an architectural model AM_1. This model AM_1 is refined into another architectural model AM_2. The Tracing Toolkit is able to identify that the requirements are now also satisfied by the components in the refined model AM_2. When checking the satisfaction of SR_4 both $Component_{3rev}$ and $Component_{4rev}$ in AM_2 will satisfy this requirement. AM_3 is yet another refinement of the architectural model of AM_2. Direct satisfaction relations can also exist towards this model AM_3. The Tracing Toolkit will be able to identify that now SR_4 is also satisfied by both $Component_{3rev2}$, $Component_{4rev2}$ and the newly introduced $Component_{5rev2}$ in AM_3.To indicate which element satisfies a requirement, the Tracing Toolkit uses both the satisfying components name and the model in which it resides. E.g. in Figure 21, the requirement SR_4 is satisfied by: $Component_3$ in model AM_1; $Component_{3rev}$ and $Component_{4rev}$ in model AM_2; and $Component_{3rev2}$, $Component_{4rev2}$ and $Component_{5rev2}$ in AM_3. The Tracing Toolkit is able to recognise these complex behaviours.

- Derivation analysis: This kind of analysis is basically the reverse of coverage analysis. It works in a similar ways, only now the main check is whether a implementation artefact actually has a requirement, from which it originated. The Forest tool and the Tracing Toolkit treat this kind of analysis in a similar way as the coverage analysis, only reversing the direction. In Forest this means following the delegation from an atom to a requirement. In the Tracing Toolkit, this means posing the same query, but letting Prolog resolve the other end of the trace relation. Here again refinement of a model is incorporated in the consistency checking.

- Change management: Change management is another kind of analysis which means that analysis is performed on the implications that a change can have for a system. Many examples of such analysis can be named; Checking which design is affected by a change in requirement, or which

requirements are affected when certain code is changed. Change management is explicitly left outside the scope of the Forest project, and not supported.

Change management, is what we have called consistency checking between requirement level and lower design level, and highly important in the Tracing Toolkit. Currently only the consistency between requirement models and architecture models is checked. In case of a change to one of the elements in these models, it is checked whether the model will contain consistency when these changed are made. When the change will imply a problem in maintaining a model consistent, the Tracing Toolkit will present the user with a choice to continue, or abort the action.

- Rich traceability: The Forest project mentions the use of storing design decisions as a rationale for taking design decisions as a possible source of tracing. (to create what they call rich traceability). It is currently only supported in the metamodel (by making it possible to store the rationale), and not implemented. The Tracing Toolkit also offers the possibility to store design rationales in requirements, and even in the model storing the traces, explaining why a trace is created. Supporting any tracing to include this rationale would require implementing additional Prolog-queries, formalizing relations to check and supporting these in the toolkit.

Kind of (automated) traceability check

Related to the previous criteria, we will here discuss how the two tools offer automated checking of the analyses and in which way.

- Implementation of requirements: The Forest tool offers automated checking of requirements implementation from the moment that a requirement is entered into the system. When a requirement is not implemented in a atom, the system will display a red stopping-sign icon next to the requirement, indicating to the user that a problem exists. The Tracing Toolkit also offers automated support for checking implementation of a requirement into design. To indicate problems with the implementation of a requirement, the Tracing Toolkit displays the requirements in red. When a requirement is correctly implemented (satisfied), the requirement will be displayed green. Next to this, is also checks the implementation of a requirement (satisfies) when modification on a requirement are performed.

- Presence of requirements for design artefacts: The same automated checking is also performed in a reversed way, checking whether design or implementation has a requirement associated with it. Both tools offer this reversed behaviour of the implementation check.

- Presence of externally referred files: The Forest tool automatically checks whether every externally referred file also actually exists. Only existence is checked, and not whether the file actually contains any useful information. The Tracing Toolkit does not offer this kind of behaviour.

- Refinement of one model (element) into another model (element): On modifying a requirements or design which is refined into a more detailed version in another model, the system can provide user assistance by notifying him of this fact. The Forest tool doesn't support this kind of refinement. The Tracing Toolkit does offer this behaviour as described in the section about coverage analysis.

Extendibility of analysis

With extendibility of the analyses we refer to the possibility to add extra analyses to the current tool support, to fulfil the goals of the stakeholders in the tracing process. The way that the Forest tools perform trace analysis is quite restricted, as the only relation between the metamodel elements is the delegation relation. This relation is currently exploited to create the traces from requirements to implementation and visa-versa. When we inspect the metamodel some extensions could be possible in the direction of tracing images or motivation, but not much more. In this sense, the metamodel, aimed explicitly at simplicity, forms a barrier.

The Tracing Toolkit is less restricted in the possibilities of analyses. This was also to be expected, as the Tracing Toolkit is designed with traceability as a goal. As a result more tracing analyses can possibly be carried out. One big advantage of treating traces as a role of other relations such as

requirement relations, architectural connectors, and trace relations, is that everything can be considered a trace. In this way, both intra level relationships and also inter level relationships can be treated as traces. Extending the tracing is also much easier. The only barrier for using new relations and using them in the tools, is defining formally when the relations are valid, forming a Prolog query to check them, and representing them in the tool. If a certain stakeholder wants to trace the number of implementation elements that a certain stakeholder caused, this is possible; it should only be implemented into the tool, when this is needed.

Another big difference between the two tools is the explicitness of the trace model in the Tracing Toolkit. This offers the possibility to type the traces, and treat each trace differently. Each type of trace can have a different formal semantic relate to it. These semantics describe the relation, and thereby define when they are valid. These formal semantics can be translated to Prolog-queries, which check the formal semantics of the relations as they are used. This is used as basis for the consistency checking.

How to perform the analysis (Automated / Manual)

The way that the analysis is performed differs in both tools. This is mainly because of the goals of the tools. The Forest tool is designed to create documentation augmented with information describing the implementation of requirement. As a extra function, is also offers the possibility to automatically check this implementation checks when inserting the information into the tool. It performs these checks on all the present artefacts in the tool. The tracing tool works similarly, performing analysis when a change in one of the models occurs and performing static analysis when a model is loaded.

7.3.3 Usability

In the area of usability, the tools differ quite a lot. This is due to the amount of time spent on the actual tooling in both projects. Building the tooling was one of the main goals of the Forest project. A student has been working on implementing, more or less stable ideas, for a large amount of time. In the Tracing Toolkit, the tooling is more a proof of concept, in which the concepts also had to be developed in the same time as the development took place. This partly explains the amount of detail, and some of the design-choices such as using (partly) generated editors.

Creation of requirements and design

Requirements are created in very different ways in both tools. The Forest tool uses a tree view like structure to edit its requirements and design model. This is integrated in one treelike structure. Entering a new requirement involves creating a tree-item, which opens a screen to add the details of the requirement. Any subsystems implementing this requirement are treated as a child of this requirement, being located directly beneath the requirement. Editing a requirement involves selecting the requirement, and editing the details of it in the detail screen. Any sub requirements are created by selecting a requirement, and creating a new requirement when the selection is active. Any other actions like deletion or moving is done using a context menu of the selected element.

In the Tracing Toolkit, requirements, traces, and architectural design are created separately from each other. Creating requirements and design involves graphically editing a model instance, by dragging elements on a canvas. Attributes of an element are set by modifying a properties view of the element. Deleting and modifying an element is done using either the context menu of the element or via the menu bar. Creating relations between elements in the same model (not traces) is quite differently from the Forest tool. Relations like decomposition are not created implicitly. Instead, relations are explicitly created by selecting a certain relation in the tool menu and dragging a line from the source to target element. This enables the creation of complex relations, and explicitly makes the relations visible. Attributes of the relation can also be set using the property view.

In conclusion there is no real difference in the usability of the requirements, and (architectural) design editors. The Tracing Toolkit offers more insight into the different relations that can exist in the models because they are created explicitly and graphically.

Creation of traces

The creation of traces is also a quite different in both tools. The Forest tool creates traces, by specifying a delegation relation. This delegation relation is also created in the same tree-view-like structure. When a new delegation is created, a source (requirement) and target element (any element) have to be specified. This relation is the basis for later on checking traceability of e.g. implementation.

The traces play a more central role in the trace toolkit. A separate editor is available to create traces. First a separate trace file needs to be created through a wizard. This wizard asks which two models need to be related by the traces. Once this is in place, a editor is presented featuring a trace matrix, with the elements of both selected models on each axis. Creating a trace is now a matter of selecting a cross-section of these axis (a cell in the matrix), and indicating which trace to be created (currently only satisfied, and refines). A checkbox is placed in the trace matrix when a trace is created.

In conclusion, the tracing editor gives more explicit insight in the traces that are created. The traces are very easy to assign, and are not dependent on any other structure (like the decomposition) like in the Forest tool.

Easy of use through non mandatory information.

In both tools almost every attribute or property of an element are non-mandatory. This eases the entering of information, as the user is not burdened with details which need to be filled in. There is no real difference between them.

Defining of (sub) systems should be easy

Defining subsystems within the Forest tool involves selecting a system, and right clicking, activating the context menu, where a new subsystem can be introduced. In this way a hierarchy of systems can be created. In the Tracing Toolkit, one simply drags a new system into the canvas, and needs to connect this system to its parent using a decomposition connector. Both interactions take little effort.

Integration with the Development environment

Both the Forest tool and the Tracing Toolkit have been designed as an Eclipse plug-in, to be integrated within the Eclipse IDE. Both are very similar in this respect. The way that the functionality is presented is however different. Also the dependencies that the both plug-ins have with other plug-ins will be different. The Tracing Toolkit, being implemented with EMF, GMF etc. uses more dependencies, and will require more plug-ins to be installed into the development environment.

Reuse of artefacts

The Forest tool has an option to create abstract systems which have requirements associated with it that can later on be inherited by a system extending the abstract system. This enables common requirements to be reused in multiple projects. This kind of reuse originates from a business need specified by Chess. The Tracing Toolkit does not include reuse, because it has been explicitly excluded, as it is outside the scope of the project.

7.3.4 Extendibility

Support for different traces to be created

This criterion is the one of the main differentiating factor between the both tools. Tracing requires relations between elements to be established. Extendibility of tracing, roughly translates to the extendibility of relations that can be created. The Forest tool is very limited in the amount of tracing that can be performed by its current structure. It relies mainly on the delegation mechanism. Not many other relations can be exploited to enable tracing of other natures.

The Tracing Toolkit offers a much wider amount of relations which can be used for tracing. As well in the requirement metamodel, as in the architectural metamodel, we have defined a large collection of

relations, distilled from literature. These relations can, be exploited for traceability purposes. Every relation available in both metamodels can be used to create custom traces.

Next to this existing relation, also interlevel and intralevel relations can be established, through the use of the trace metamodel. The trace metamodel is extendible with custom trace relations types. This enables the extendibility of tracing between models.
Because every artefact in the all the models of the Tracing Toolkit are translated to Prolog, we can reason about these models using Prolog. We can state queries to for example enable a new way of tracing. Currently only tracing between requirement and architectural level is supported. It is however possible to extend this to lower level design phases. This requires creating metamodels and editors for these design levels and integrating these into the Tracing Toolkit. A formal definition then needs to be specified for the relations between these models. Once this new type of trace is added to the trace editor, Prolog queries can be used to check properties of the new models. Ultimately arriving at checking the consistency the lower level designs (or code).

Support for different editors

Both tools feature an open (XML/XMI) based repository, on which editors can be built. As a result, both tools offer the possibility to extend the editors to support certain stakeholders. Because the Forest model integrates the storage of all data, in one repository, it is almost mandatory to also create one editor for modifying it. The Tracing Toolkit separated the repositories per development level. This makes it easier to create custom editor to support each development level in the best way. The current requirement editor could fore example easily be replaces with a tree-view-like or text based editor.

Generation of documentation

Generation of documentation is currently only supported by the Forest tool. This is a direct result from the goals of the project in which they are created. The Forest-project has stated document generation as one of the goals, and the project of the Tracing Toolkit has not. This does however not mean that it is not possible. All information stored in the models of the Tracing Toolkit can easily be transformed to another representation, such as documents. All what is needed is to add a model-to-text transformation to enable the generation of documentation, based on the repositories of the Tracing Toolkit.

Level of detail in modelling

The amount of detail in the metamodels differs quite a lot between the Forest tool and the Tracing Toolkit. One of the goals of the Forest project, to maintain simplicity, is directly reflected in the detail level of the metamodel, offering less modelling constructs. The only mechanism to increase the level of detail in the Forest repository is the ability to decompose systems into subsystems and ultimately atoms.

The project, in which the Tracing Toolkit is designed, is aimed at experimenting with state of the art in maintaining consistency between requirement, and architectural level. As a result, there is more time spent in analysing the current state of the practice of requirements and architectural modelling. This is reflected in a more extended metamodels, which offer more modelling constructs. This enables a user to capture more information in the model instances.

Freedom in Relations

The amount of freedom in creating relations between model elements is related to the level of detail in the metamodels. The Forest tool is here again very limited. The Tracing Toolkit offers much more possible relations to be created. Not only the on intra level, but also in the inter level relationships. Because the modelling constructs for creating relations are much richer, also the possible usages of these relations in trace analyses is much greater, improving the possible advantages of tracing.

7.3.5 Scalability

In this section we will discuss the scalability of the editing possibilities. The scalability of the storage will be left outside the comparison because these differ very little, both being based on XMI/XML.

Requirements editing

The editing of requirements is very different in both tools. As a result, the tooling also scales very differently. The Forest tool uses a tree-view-like editor to edit the requirements. The tree containing the requirements has a parent/child relation with sub requirements through a decomposed mechanism. This offers the ability to hide a certain level of decomposition. It also offers the possibility to filter out certain tree nodes, to improve the overview. The tree-view-like structure resembles most requirement documents, as these are also often built up according to a decomposition of requirements into sub requirements. As a result, the editor of the Forest tool is reasonably scalable.

The Tracing Toolkit on the other hand, features a graphical editor to edit requirements. The advantage is that is offers insight into the complex relations that can exist between requirements. The downside is that scalability is compromised as the amount of graphical elements gets bigger. The tool offers a zoom function to enable “the big picture” to remain in sight. We can however not ignore the fact, that editing big requirement models in this way is not very suitable. This is a downside of choosing a generated editor, against building a custom designed editor. It is a matter of further research to improve the editing of requirements, to better suit scalability and also insight into the complex requirements relations that can exist.

Design editing – subsystems in Forest

Editing design in the Forest tool is performed in the same way as editing requirements; Through the use of a tree-view-like editor. Again the same arguments on scalability again hold, as were stated in the previous section. A big downside in this respect is however, that design is often not represented well in tree-structures. Often a designer uses a boxes and lines approach to make a high level design. Because a high level design does often not contain very large amount of elements, a graphical way of editing is here very well suited. In this respect, the Tracing Toolkit offers a better suited way to edit design. Scalability is often less of an issue. If a design does become very large, the same problems again hold for the graphical editing.

Representation / Creation of traces:

Earlier, we already established that both tools treat traces quite differently. This is also reflected in the creation of traces, as discussed in the usability criteria. When looking at this difference from a scalability perspective, we can say that there is also a big difference. The Forest tool creates traces through its delegation mechanism, which needs to be explicitly specified, per requirement (A requirement is delegated to (sub) systems). These systems will ultimately be connected to atoms. Delegations are represented as separate nodes in the tree-view-like structure. Per delegation, a source and target needs to be chosen (represented as a child). This involves quite a lot of steps, per delegation. When a large number of delegations are present, this will eventually be not very insightful anymore. The user of the tool will easily get lost in the large number of delegations that need to be inspected closely to see which elements they relate. Because delegations are represented in the same tree with the requirements and design, the overview is quickly lost. To partly alleviate this problem, filter can be set on the tree-view, to for example only show delegations or requirements.

Next to creating traces (or delegations), in the tree-view, the Forest tool can also represent the traces differently. A special trace view is present, which can show a trace from a requirement to an implementation, or the other way around. An example can be seen in Figure 33.

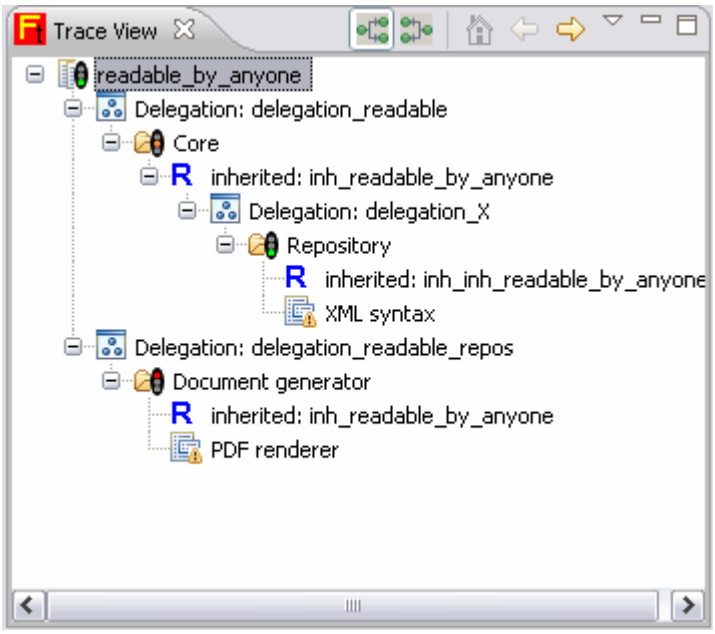


Figure 33: Forest trace view

Here a requirement, reasable_by_anyone is traces to two implementations, being XML syntax, and the PDF renderer. The readable_by_anyone requirement is inherited by the Core component, the repository and the document generator, which both implement them in a separate way. The systems are the Core, Repository and Document generator. The atoms here are the XML syntax and PDF renderer.

Lastly, the tool also offers the possibility to generate a Trace Matrix to represent the traces, present in the system (see Figure 34). This shows which requirement is implemented by which other component in the system.

	S: Repository	A: Plain Old Java Objects	A: XML syntax	S: Model checker	A: ModelChecker	A: CheckDelegator	A: Checks	R: multiple_checks	R: java_implementation
R: java_implementation	X	X							
I: readable_by_anyone	X		X						

Table 2: Traceability matrix of Core

Figure 34: Generated Trace Matrix in Forest

The Tracing Toolkit takes a different approach by using a Trace matrix as the input, as the editor for creating and representing traces, is similar to a trace matrix. This matrix represents the elements of the different models on both axis, and when a trace needs to be created, the desired kind of trace is selected in on the cross sections (see Figure 35).

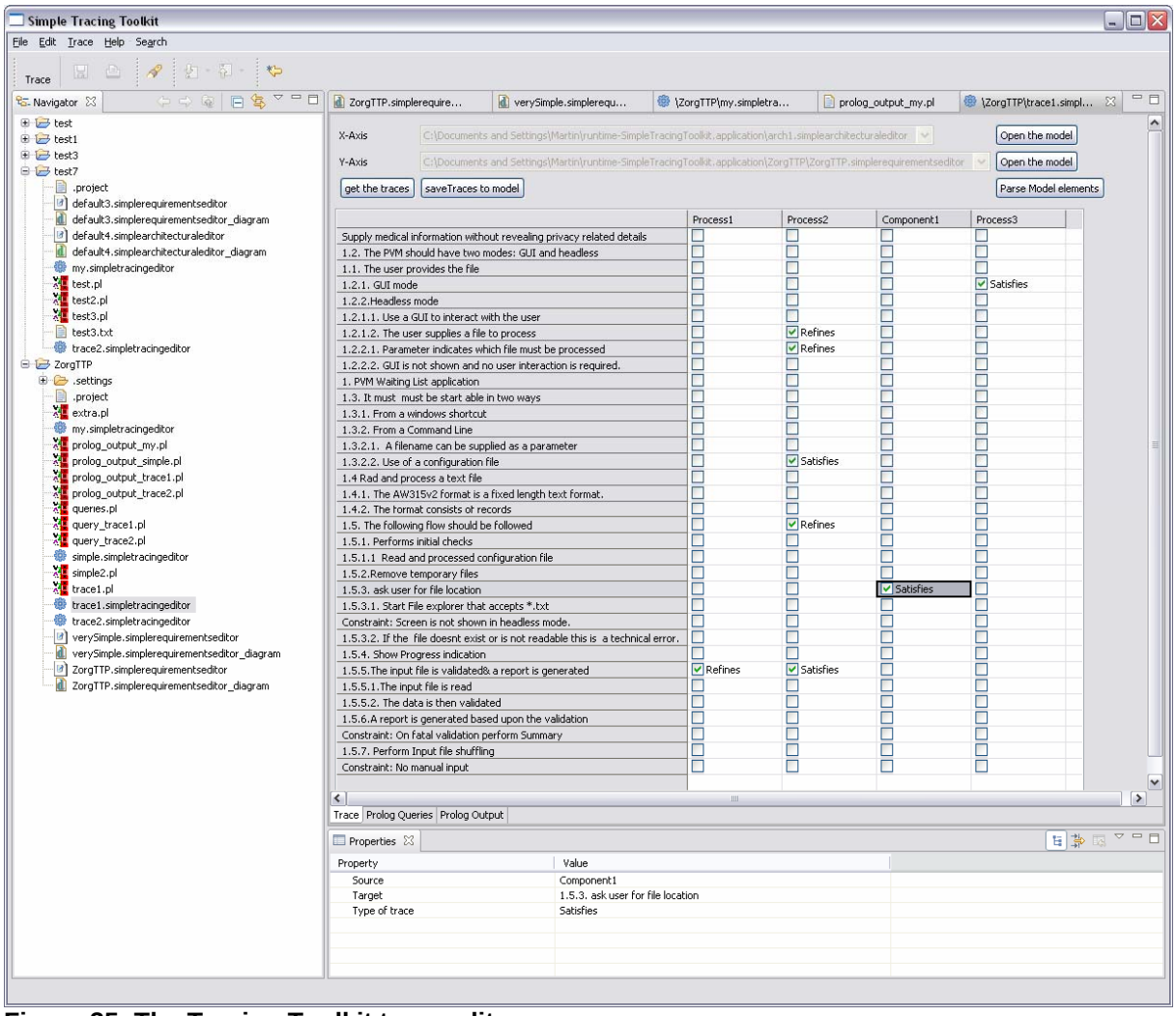


Figure 35: The Tracing Toolkit trace editor

The overview of such a trace editor can be compromised when a very large number of traces are represented in the editor. A possible extension to solve this scalability problem would be to use filters on the axis of the editor. The overview of which traces are created is however more visible than in the tree-structure, used in the Forest tool.

7.4 Conclusion of the comparison

In this section we will present the conclusions on the comparison between the Forest tool and the Tracing Toolkit. These conclusions will provide an overview of which tool excels at what specific comparison criteria. We can only draw conclusions on which tool is best in what respect, in those criterions which they both satisfy. We will discuss these conclusions per category of the comparison framework. This will result in a summarizing table, presenting the result of the comparison.

7.4.1 Storage

Storage wise it is hard to pinpoint what tool offers the best possibilities. The location of the storage and the openness of the storage format differ very little. The main difference in storage lies in the fact that the tracing tool uses separate files to store each instance of the metamodel, instead of one big file containing all the information. This separation of concern can be an advantage when (parts-of) the metamodel needs to be extended. The impact of changing one of the metamodels in the Tracing Toolkit will have a more limited impact instead of having to change the metamodel of the Forest tool. Changes will in the Forest toolkit be limited to just the parts of the application that edit and use the metamodel. The rest of the application will not be affected.

7.4.2 Trace checking

An area where the biggest differences between the two tools can be found is the way that they handle traces. First, the creation of traces is more explicit in the Tracing Toolkit, where one has to specifically specify what trace relations exist, and what kind. In the Forest tool, this is implicit, by decomposing the system from requirement to implementation. Explicit or implicit creation of traces is a matter of choice, and there is no real good or wrong here. The representation of creating the traces does however become clearer when one specifies the relations explicitly by checking a trace matrix.

The kind of analyses that can (possibly) be performed is an area where the Tracing Toolkit has more possibilities. Because it is designed with creating traces as a primary means to maintain consistency, it has as a result more expressiveness in this area. The Tracing Toolkit offers a separate trace metamodel which contains typed trace relations with formally defined semantics. These semantics are used to implement the Prolog queries, used to check the consistency of the models. By formally defining the relations, the Tracing Toolkit performs more sophisticated trace analyses. Currently, the same kinds of analyses are performed by both tools. The Tracing Toolkit however performs a more sophisticated analysis of the implementation of a requirement. It can incorporate checking on transitive refinement of design models, satisfying a requirement. This is an example of a more sophisticated analysis, not possible in the Forest tool.

The extendibility of the analysis is better in the Tracing Toolkit. A greater number of trace types can be specified, which can later-on support a larger amount of analyses. As a result, the extendibility of the analyses to perform is better. The way that the analyses are performed differ very little, both offering dynamic and static analysis.

7.4.3 Usability

The Usability criterion is difficult to judge, because there is no good and wrong. Usability needs to be proven when the tools are used in practice (e.g. using a case study). We can see, that the tree-view-like requirement editing that the Forest tool offers looks better suited then the graphical editor included in the Tracing Toolkit. This is because the tree-view-like editor better resembles the natural decomposition, often found in requirement documents, than a graphical one. When requirement documents are simple and straightforward, this will certainly be true. When complex relations between requirements need to be represented, this will become problematic. It is currently not possible to assign any other relation than the decomposition. This is caused by the fact that tree's are ideal to support one relation structure, but not for including other (inter-leaf) relations. The graphical editor of the Tracing Toolkit offer a rich set of relations, offering possibility to graphically depict them. This enables it to represent more complex requirement documents.

Looking at the design editors, it becomes clear that the approach taken by the Forest tool to represent design is not well suited. Designers often create design in a graphical way, using boxes and lines diagrams. These diagrams have to be translated into the System/Sub system structure, used by the Forest tool. The Tracing Toolkit offers graphical editing possibilities, offering more intuitive editing for architectural designers. It offers more support for modelling complex design elements. The Forest tool, only features a system/sub system structure.

Editing of traces is another part which is hard to judge. The implicit created traces in the Forest toolkit suffer from a problem in presenting the overview. Mainly because each delegation has to be inspected individually to see which elements it connects. The big picture becomes clearer when all the created traces are presented in one graphical representation, as done in the Tracing Toolkit. Lastly, creating these traces in the Forest tool is troublesome. Trace relations are created by creating a delegation. This takes a number of steps to create. Creating traces in the Forest toolkit implies more work, then selecting a checkbox in a Tracing Toolkit.

7.4.4 Extendibility

In the area of Extendibility, both tools have possibilities. Currently the Tracing Toolkit has more elaborate metamodels to allow traces to be created. This offers the possibilities to extend the tracing analyses. In the area of extending the metamodels, both tools have possibilities. Extending the

metamodel in the case of the Forest model would require a more drastic change to the entire system, because only one metamodel is available. The Tracing Toolkit offers the possibilities to change the metamodel per development phase. Changing one metamodel only affects a part of the tool.

The Tracing Toolkit has a separate trace metamodel, offering typed trace relations with formally defined semantics. This metamodel has built in extension mechanisms. It is as a result possible to extend the kind of trace relations of the Tracing Toolkit. Tracing possibilities can easily be extended by introducing new trace relation types. They need then to be formally defined, so that properties of these relations can be checked. The Prolog engine can be used to check properties of these newly created relations, possibly using them to extend consistency checking. Lastly, the amount of constructs offered in both the requirement and (architectural) design models are much richer in the Tracing Toolkit, than in the Forest tool. These constructs can be used to extend the possible tracing. (E.g. trace a stakeholder to design elements).

7.4.5 Scalability

The scalability of the requirements editor embedded in the Forest tool is better to suited to represent large, simple, decomposition relation requirement documents. The graphical editor used in the Tracing Toolkit is less scalable to the number of requirements. It has however the advantage that is can represent more complex relations and structures. These relations cannot be represented in the Forest tool, limiting the kind of requirement documents that can be represented.

The design editor in the Forest tool has an issue with representing big system structures. The reason for this problem is that trees are not the most suitable representation mechanism for architectural design. When the number of design elements gets bigger, the big-picture of a design is quickly lost. The Tracing Toolkit, offering a graphical editor to represent architecture has a more intuitive way of editing design, and is in this respect better suited. It also again offers more constructs to represent complex structures.

Editing and representing traces is a highly immature domain in software, engineering, and it is difficult to say which of the two tools performs best. The creation of traces through the delegation mechanism looks to be problematic in this respect; each delegation involves a source and target artefact to be linked. When systems become bigger, the number of elements to relate also becomes bigger. The amount of delegation relations will grow very quickly. The representation of traces in a trace matrix is a more scalable approach, because it only involves checking a box to create a trace. The size of the matrix does however grow very large when large models are related. Investigation into filter will need to be done in order to solve this. Filters are also included in the Forest project, as a way to improve oversight in the number of elements in the tree-view.

7.4.6 Comparison table

We have summarized the result of this comparison in a graphical way in Table 4.

Criteria	Forest tool	Tracing Toolkit
Storage		
Storage location	Project folder, containing Eclipse resources.	Project folder, containing Eclipse resources.
Kind of information stored	One file	Storage per metamodel instance.
Openness of storage format.	XML	XMI
Extendibility of the storage format.	Changing DTD of the metamodel.	Changing metamodels, isolated per metamodel.
Trace checking		
Creation of traces	Implicit, by delegation	- Explicit, stored in a separate trace- model - Treat normal relations as trace-candidates

Criteria	Forest tool	Tracing Toolkit
Kind of analyses to perform	- Coverage analysis - Derivation analysis	- Coverage analysis (using transitive refinement) - Derivation analysis - Change management
Kind of (automated) traceability check	- Implementation of requirements - Presence of requirements for design artefacts - Presence of externally referred files:	- Implementation of requirements - Presence of requirements for design artefacts Implementation of requirements using
Extendibility of analysis	Restricted to the delegation relation	Highly extendible by offering - Separate trace model - Types trace relations - Formally defined trace relations, implementable in Prolog queries - Treating within-model relations as traces
How to perform the analysis (Automated / Manual)	- Dynamic - Static	- Dynamic Static
Usability		
Creation of requirements and design	Tree-view-like editor	Graphical editor
Creation of traces	Creating delegations in the tree-view	Graphical trace matrix
Easy of use through non mandatory information. (Use as see fit) Defining of (sub) systems should be easy	Possible	Possible
Reuse of artefacts	By offering abstract systems	Not supported
Extendibility		
Support for different traces to be created	Limited to delegation	Highly supported
Support for different editors	Creation of a different editor to edit the repository is possible	Different editors to edit each metamodel instance is possible
Generation of documentation	Non-Template based generation is implemented	Future extension.
Level of detail in modelling	Low	High
Freedom in Relations	Low	High
Scalability		
Requirements editing	Good	Problematic
Design editing – subsystems in Forest	Problematic	Good
Representation / Creation of traces:	Limited	Limited

Table 4: Comparison Summary

7.5 Summary

This final chapter discussed the comparison of the Forest tool with the Tracing Toolkit. In order to make a founded comparison, we have described our comparison framework. This is based upon the requirements of both tools, general tooling criteria (usability, extendibility and scalability) and some criteria specific for tracing. Before comparing the tools, a description of the Forest tool is given. We discussed the structure of the repository, by explaining the metamodel of the tool. Also the actual functioning of the Forest tool is described.

The comparison compared the two tools on the area of; Storage, Trace checking, Usability, Extendibility and. Scalability.

Storage wise, the only difference is that the Tracing Toolkit separates the storage of each type of model. This enables us to change the tool with lesser impact on the general application-structure.

The biggest difference lies in the area of trace checking. The Tracing Toolkit, offering an explicit trace editor, improves the general overview in a development process. More importantly, the Tracing Toolkit offers typed traces, with formally defined semantics for the traces. This allows these relations to be used for checking consistency. The Forest tool does not offer typed or formally defined trace relations. The Tracing Toolkit furthermore offers more ways to extend the analyses of the tracing, than the Forest tool. In the area of usability it is proved difficult to favour one of the tools. The requirements editor of the Forest tool looks more usable because it has had more development time. It is however very limited in the constructions it offers to model a requirement document. The graphical requirement editor of the Tracing Toolkit offers a better insight into complex relations. The design editor of the Forest tool looks subordinate to the graphical way of modelling designs in the Tracing Toolkit. The reason is that design is already often modelled graphically. In the area of trace editor, the Tracing Toolkit has an advantage of explicitness. It is also faster to create a trace in the Tracing Toolkit than it is in the Forest tool.

In the area of Extendibility both tools offer possibilities. The Tracing Toolkit has an explicitly defined metamodel, which has built in extension mechanisms. The Forest tool does not, and as a result is less favoured. Extending the analyses that the Tracing Toolkit can perform is possible. It requires extending the trace metamodel with other relations, formally defining these relations, and using these semantics to check certain properties of the relations (such as the *satisfies* relation is used to check consistency).
In the are

The major difference in the area of scalability lies in the scalability of the editors. The requirement editor of the Forest tool is more scalable towards bigger models. The architectural editor of the Forest tool also has scalability issues.

The end of the chapter presents the conclusions of the comparison in a comparison table

8. Conclusions

8.1 Introduction

The goal of this thesis is to investigate whether we can improve the maintenance of consistency between requirements and lower level design artefacts in a software development project, in case of change. Specifically, we want to know whether this consistency can be improved by the use of techniques from Model Driven Engineering. This is formulated in the following research questions:

How can we improve the consistency between requirements and lower level design artefacts in a software development project in case of change?

1. Is it possible to model requirements and architectures using Model Driven Engineering techniques, *e.g. by using metamodels?*
2. Is it possible with these metamodels of requirements and architectural designs to enable *consistency checking?*
3. Is it possible to maintain consistency between requirements and lower level design artefacts in *case of changes?*
4. Is it possible to maintain consistency between requirements and architectural design with tool *support?*

In order to answer these questions, we will first give a summary of the previous chapters, after which we give answers to the questions derived from the main research question. Together, the answers to these questions will give an answer to the research question stated above. Lastly we will give an overview of future research and directions for improvement in this research.

8.2 Summary

In Chapter 2, we have investigated the current state of the practice in requirement modelling. We identified the most important factors that drive the modelling of requirements. The main motivations for modelling requirements include:

- It concentrates information in the requirements phase in one location, preventing scattering of information;
- The use of models reduces the communication and management problems when having heterogeneous requirement documents. This increases software quality and lowers costs;
- With the general tendency in software engineering to raise the abstraction level, models of software become more important. Only when a good requirement model can be devised, structured connections to design becomes possible.
- The growing attention to Model Driven Engineering (MDE) which, using models as input, transforms models into more detailed models. Models in MDE have the same status as objects in Object Orientation. For MDE to be fully employed, requirements should be available in models.
- The inevitable presence of change in a development process requires the need for a common language to communicate these changes. A common metamodel for requirements can be used as such, to communicate the changes in the development process.

Having established the need for requirement modelling, we identified a number of research sources, investigating the domain of requirement modelling. The work by [Vicente'07] is discussed because of its attempt to shift the requirement specification process from a document oriented one towards a model based process. It identifies the use of requirements in a MDE context as the main driver behind modelling requirements. An important contribution of this paper is the metamodel, called the REMM metamodel to capture requirements information. We have identified the main attributes and elements of this model, for possible use our own metamodel. Our focus in discussing the REMM metamodel lies in its relationships. The relationships are aimed towards the support of the Requirements Engineering (RE) process including: elicitation, negotiation, validation and documentation. The relationships are split up into two types: inter and extra requirements traceability relations. The former being specified between requirement entities, and the later between a requirement and other non requirements

entities (such as stakeholders, files etc). A full discussion on the relations (and other elements) found in this work is given in Section 2.2.1.
A second source on requirements modelling is a paper by Marshall. [Marshall'03]. It presents an overview of the basic concepts of model based requirement engineering, aimed specifically at the web based business-to-business applications. In this paper a conceptual model is presented to manage requirements. Its main contribution lies in the fact that the approach claims to manage requirements of various kinds. To achieve this, a conceptual model is used, that is mapped onto the various heterogeneous requirement documents. The authors have integrated this conceptual model within a proposed metamodel. To do this, the conceptual model is coupled to a requirement metamodel. They propose the use of a transformation mapping from each requirement document to this conceptual model. Once the mapping is in place, a change in one of the requirement models can be propagated to the other models, using the conceptual model. Effectively, this achieves a form of intra requirement-level consistency-checking.
The last source on requirement modelling that we have discussed is the SysML language [OMG'04]. It offers modelling constructs to represent requirement diagrams. It specifies a number of relationships between requirements. No new constructs have been discovered from this source.
Having discussed the existing work in the area of requirement modelling; we have established criteria for selecting model elements in our own metamodel. These criteria are found in Section 2.3.1. Using these criteria, we have selected the following basic requirement concepts (see Section 2.3.2).
- Requirement: A condition or capability needed by a user to solve a problem or achieve an objective [IEEE-610'90;
- Stakeholder: People or organisations who will be affected by the system and who have a direct or indirect influence on the system [Sharp'99];
- External Object: An entity related to a requirement, capturing details of the requirement;
- Rationale: A rationale is the reason why a requirement is introduced.

The following taxonomy of requirements is used: (Section 2.3.3)
- Business Requirement: A requirement originating from the business domain;
- Software Requirements: A condition or capability of software, needed by a user to solve a problem or achieve an objective;
    - Functional Requirements: A software requirement describing what the software should do;
    - Non-Functional Requirements: Describes a quality attribute of a Functional Requirement.
        - Constraint: A Non-Functional requirement describing mandatory quality attributes;
        - Property/Value pair: A combination of two coupled values, used to capture the values of Non-Functional quality attributes.

The following relations between requirements are used (Section 2.3.4)
- Dependency: A relation indicating a specified dependency-type relation between requirements;
    - Requires: a type of dependency relation between two requirements, indicating that one requirement is needed to fulfil the other;
    - Excludes: a type of dependency relation between two requirements, indicating that if one requirement is chosen, the other requirement in the relation can't be chosen;
    - Depends: a type of dependency relation between two requirements, indicating a dependency that can not be captured by the above;
- Refine: A relation between two versions of requirements;
- Influence: A relation between two requirements that have a cost or priority influence to each other;
- Decomposition: A relation between a part and the whole;
- Affects: a relation between a software requirement and a Non-Functional Requirement.

A full description of the proposed metamodel can be found in Section 2.3.5. A discussion of the difficulties and possible extensions to this model can be found in Section 2.3.6.

In Chapter 3, we investigated the modelling of architectural design. We have established that in the research community, there is a lack of consensus on the definition of architecture. We have chosen a definition by [Bass'03], that has the following characteristics: It models the interaction of elements; it describes more than one structure; it defines that every software system has an architecture (implicit or explicit); the behaviour of the system needs to be captured. Next we have identified the need for modelling architecture as:
- It enables the communication amongst stakeholders;
- It represents early design decisions;
- It offers a transferable, reusable abstraction of the system.

We have furthermore identified that designing an architecture is a task in which multiple forces on the architect are involved. A feedback loop can be detected in these forces, making it a difficult task. We

have also discussed the ability to express an architecture in multiple views, aimed at specific stakeholders in Section 3.1.3. Next we have discussed the different architectural development methods that drive the creation of an architecture in 3.1.4. We have discussed RUP and Synbad. Next we have given a discussion on Architecture Description Languages (ADL). We have identified the key components of an ADL, according to a classification and comparison framework discussed in [Medvidovic'97]. (Section 3.2) We have used this framework as a common denominator on what an architectural model should contain. The architectural metamodel as a result contains the main elements found in the framework: Components, Connectors and Configurations. We defined criteria to restrict the elements that should be contained in an architectural metamodel. We established the following criteria:

- We must offer (only a limited) support for evolution/versioning.
- We must offer only a limited support for evolution/versioning.
- We must offer support for the management process of a system.
- We must offer support for the dynamic (runtime) interaction of architectural elements.
- We must offer support for decomposition of a system.
- Interfaces and constraints of components and connectors must be able to be specified.

Contrarily to some other approaches, reuse of models is not our main concern.
According to these criteria, we have derived an architectural metamodel, inspired by the elements in the ADL classification framework and the discussed literature. We have included both static and dynamic (runtime) elements in the metamodel. The following relations are used in the metamodel:

- Allocation relation: a relation between an element of the system and elements representing artefacts outside the system. Specifically aimed at supporting the management process of the development project;
- Process relation: a relation between dynamic components of the system;
- Module relation: a relation between static components of the system;

In Chapter 4 we focussed on trace relations, and proposed a metamodel to capture trace relations. To define what a trace is, we have discussed definitions on traceability and described why people are interested in traceability. We have concluded that the concept of traceability is still very much in its early phases. As a result, the meaning of traceability differs per stakeholder. We have identified that besides the goal of a trace, the elements which the trace relation connects also highly influences the meaning of the concept of traceability. These elements can differ in granularity and in the attributes that they capture. Another contributing factor to the difficulty of defining traceability is the fact that throughout the industry, different relations are being treated as traces. We have given an overview of the kind of relations that currently exist, the directions of the trace, the attributes of a trace, the way that trace relations are created, and how to represent traces.
We concluded that there is little consensus on the terminology used to differentiate relations in literature on tracing. The main issue is that the term 'abstraction' can be used to represent different thing. We have therefore defined our interpretations of the terms *interlevel* relation, *intralevel* relation and *within-model* relation. An *interlevel* relation is a relation between model elements and/or models in different development phases within the same iteration of a software development process. An *intralevel* relation is a relation between model elements and/or models in different iterations within the same development phase of a software development process. A *within-model* relation is a relation between elements in a model in one single iteration and in one single development phase of a software development process.
Because the concept of a trace relation is very dependent on the context in which it is applied, we will treat a trace as a role of relation. This enables us to treat interlevel relations, intralevel relations and within-model relations as a trace. Within-model relations are already in place, as described in both the requirement and architectural metamodel. We need a way to define intralevel relations and interlevel relations. For this end, we have designed a trace metamodel which enables us to relate two MOF-based models using a special relationship, called a traceLevelRelationship. We have specialized this relation to contain two types of trace relations: *satisfies* and *refines*. The *satisfies* relation is an interlevel relationship, connecting a requirement model to an architectural model. The *refines* relation is an intralevel relationship, connecting two models on the same level of development, but within different iterations, possibly adding details. The *satisfies* relation can be used to check the consistency of a requirements model with respect to lower level models. We discussed the term consistency in Section 4.7. We defined a model to be consistent when every element in a model M1 is satisfied by (a set of) element(s) in a model M2. M2 can furthermore be refined into another model 2. The definitions of consistency of a model with respect to another model are found in Section 4.7. Because we focus on interlevel consistency, the target and source model will be at different development phases.

Chapter 5 describes the validation process of both the requirement and the architectural metamodel. To validate the requirements metamodel, we have used a case study delivered by Chess. It involves a requirements document of a software development project that has been carried out at chess. In order to validate this (rather large) document, we have first mapped the document onto an intermediate model. This intermediate model contains the same structure as the metamodel, but is in the form of a flat text document. This document now contains the elements of the original requirements document, but is tagged with the attributes of the requirements metamodel (on which the elements need to be mapped). We have partially mapped this intermediate model onto the actual metamodel. This shows that a mapping can be made. The rest of the validation has taken place on the intermediate model. All the elements in the original requirement document could be mapped onto elements of the metamodel, by which the validation is successful. The results of the validation can be found in Section 5.2.3. We have also validated the architectural metamodel. Because there was no architectural design available for the requirements validated earlier, we decided to take another design. We have used the work of [Noppen'07], in which a high level architecture for a traffic management system is given. We have performed a 1:1 mapping of this design onto elements of our architectural design. We found that every model in the design could be mapped to an element in the metamodel. As a result, this validation was successful. The result of this validation process can be found in Section 5.3.3.

In Chapter 6 we explain the tool support in order to support the maintenance of consistency between requirements and architectural design. The tool support is built on top of the Eclipse Rich Client Platform (an application platform for building generic applications). An RCP application is a collection of components, called plug-ins which cooperates to deliver the required functionality. The advantage is that it is easy to develop an application which can be used as a plug-in for the Eclipse Java IDE (basically also an RCP application). By developing our application as an Eclipse plug-in, we can load the application as a plug-in of the Eclipse Java IDE, which makes the tool easy to use for people using the Eclipse IDE, such as application developers. This integrates the consistency checking in the development process. Our tool support is highly based on metamodels, as our primary structure for saving requirements, architecture and traces. Using the Eclipse Modelling Framework (EMF), and Graphical Modelling Framework (GMF), we have generated editors for both the requirements and architectural models. We furthermore built a custom editor (using functionality of EMF), to edit the trace model, effectively creating traces between models. This editor enables the user to create relations between two model instances using a matrix representation. The tool support is currently limited to only create the *satisfies* relation and *refines* relation between two model instances. A *satisfies* relation is created between elements of two different types of model instances. A *refines* relation is always established between elements in two similar types of models. Once the trace relations are established, the tool generates a Prolog program out of the information available in both the related models and the actual traces. This Prolog program is subsequently run every time a modification to one of the related models is made. The Prolog program checks whether the modification (or change) will affect the consistency of the model with respect to either satisfaction or refinement. This supports the maintenance of consistency between the models. This chapter also offers a discussion on the strengths and weaknesses of the tooling. The strong points of the tool are the following:

- The tool is highly based on models and metamodels and uses generated code and editors;
- The tool combines multiple programming paradigms, taking advantage of both of them
- The tool offers consistency checking using formally defined semantics
- The tool is highly extendible

The tool also has some weaknesses identified as:

- The editors suffer from scalability problems when using large models
- The generation of the editors is still in its early passes, reflected in the editors that we had to use.

The future extensions of the tool are reflected in the future work section of this chapter.

The final chapter makes a comparison between our tooling and an existing tool developed by Chess in the context of the Forest project. We present a comparison framework based upon the original requirements of the Forest tool, requirements of our own tooling, general criteria for tooling, and criteria specially aimed at tracing. The Forest tool uses a very simple repository to store all its design information. The repository comprises of requirements, systems and atoms. These elements are connected via a decomposition mechanism called delegation. A requirement is in this way delegated to a system that implements the requirement. A system can also be delegated to other (sub) system (decomposing the system). A system is ultimately delegated to a so called atom, representing the

implementation level (or code). The only tracing that is available in the Forest tool, is the ability to check whether requirements are implemented (in systems or atoms), and whether a system or atom, has originating requirements. To check this, they use the delegation relation. The tool of the Forest project features an editor to edit the requirements, systems, and atoms in a tree-view-like structure. It also offers a model checker, checking whether every requirement in the model is actually implemented. The Forest tool features a document generator, able to generate a complete requirement document form the repository. The actual comparison between the tools is presented in Section 7.3. The main difference between the two tools can be summarized as:

- Storage: The Forest tool uses a single repository. Our Tracing Toolkit offers separation of models, independently saved and modified. This enables the independent evolution of each metamodel.
- Trace checking: The traces in the Forest tool are limited to the decomposition relation. Out tooling offers richer semantics in the area of tracing by offering typed traces using formally defined semantics. By using a separate trace metamodel, a wide area of traces can be established. It is possible to treat normal relationships, already found in the metamodels as traces. The trace model is clearly more aimed at taking advantage of future trace relations to be implemented. The Tracing Toolkit furthermore offers more ways to extend the analyses of the tracing, than the Forest tool.
- Usability: The Tree-view like requirements editor of the Forest tool has an advantage in the area of ease of use. This is mainly due to the larger amount of time spent on the editors. It has however very limited possibilities to represent relations in a requirement model. Only the decomposition relation is used. The Tracing Toolkit offers a richer set of relations to model the complex (within-level) relations found in requirement models. The design editor of the Forest tool has poor usability because it represents a design in a tree-view like manner. The Tracing Toolkit, offering a graphical editing looks superior in this respect. Lastly, the choice between implicit modelling of traces versus explicit modelling is difficult to judge. Creating and maintaining traces in the Forest tool is more time consuming than in the Tracing Toolkit. On each of these areas a case-study should offer proof for these statements.
- Scalability: The scalability of the requirements editor in the Tracing Toolkit suffers from scalability issues when handling large models. The same is true for the design-editor in the Forest tool. Different editing possibilities need to be investigated to solve these scalability issues.

8.3 Research Questions and Answers

In this section, we will give an answer to our main research question, by answering the questions derived from the problem statement.

8.3.1 Model Driven Engineering Techniques

Is it possible to model requirements and architectures using Model Driven Engineering techniques, e.g. by using metamodels?

Model Driven Engineering involves transforming models into other models. In MDE, this transformation could eventually result in source code, being the lowest level of abstraction. We however do not aim at (automatically) transforming requirement models into more specific models at lower abstraction levels. (The level of abstraction here, refers to the development phase of a software development project) We assume that the models in each development phase are already present. Not as a result of an automated model transformation process, but as a result of a (manual) development process, transforming the requirements via a process of design (or problem-solving) into a software implementation (See 3.1.4).

We have used the way of working of MDE, using metamodels as a key concept for the transformation of models, as a mechanism to check consistency. To do this, we used separate metamodels for each level of development. The metamodels enable us to establish relations between the development levels.

The metamodels we derived from a number of research sources are all instances of the meta-metamodel MOF (following the way of working of the MDE). As a result, relations between levels of development can be created as relations between MOF-instances. This is possible because every element in each metamodel conforms to the general MOF metamodel. This enables us to create a range of relations between the metamodels and treat them as traces. We have limited our number of

relations we treat as trace relations. We only treat the *satisfies* relation and *refines* relation as being a trace. A description of these relations can be found in Section 4.5. These trace relations can be used to check certain properties of the model and its elements in the development level.

These properties of the elements can now be combined to check the consistency between models. We have defined a requirement model to be consistent with respect to an architectural model when a *satisfies* relation (defined in Section 4.5) can be established between the two models. This means that every requirement must be satisfied by a (group of) element(s) in the architectural model. This also makes it possible to check the consistency of the architectural model by checking whether each architectural element is derived from a requirement.

8.3.2 Model requirements and architectural design

Is it possible with these metamodels of requirements and architectural designs to enable consistency checking?

In order to enable consistency checking between requirement and architectural design, a first condition it that there are stable metamodels for both development levels. Only when these are available, consistency checking can be performed. In this thesis, we have derived metamodels for both requirements and architectural design.

These metamodels have been derived from a number of independent research papers covering requirements and architectural modelling. The requirements metamodel is based upon the work of [Vicente'07], [Marshall'03] and [OMG'04], representing resent research papers in requirement modelling. In composing the requirements metamodel we have explicitly stated that we only include elements related to the structure of the requirements, and its relations to other elements in the requirements engineering process. Elements and relations to artefacts in different design levels are excluded.

The architectural metamodel is derived out of the basic components, identified by the work of [Medvidovic'97], discussed in Section 3.2. This work treats an Architecture Description Language (ADL) comparison framework. Because this work represent a common denominator of what ADL's should contain, this is a good reference for deriving an architectural metamodel. In composing the architectural metamodel, we have limited ourselves by stating a number of criteria to which the metamodel must conform: It is not aimed at reuse, has limited support for evolution, has support for the management process of the system, has support for dynamic interaction of architectural elements, has explicit support for decomposition and supports the modelling of interfaces and constraints (as described in Section 3.3.1).

The metamodels used to store the information in the requirements and architectural design phase, can be found in the Sections 2.3.5 and 3.3.2. These metamodels are validated using case studies from the software development practice. A description of the conclusions of these validations can be found in Sections 5.2.3 and 5.3.3. Our conclusions are that both these metamodels can be used in the case-studies we performed.

8.3.3 Consistency between requirements and lower level design

Is it possible to maintain consistency between requirements and lower level design artefacts in case of changes?

We have proposed a technique which uses a separate tracing model to store interlevel relations, intralevel relations and within-model relations. These trace models conform to a trace metamodel, described in Section 4.6. The trace metamodel allows us to specify typed trace relations and add formal semantics to them. Two types of relations are specified in the trace metamodel; the *satisfies* relation and the *refines* relation. The *satisfies* relation is an interlevel relation, and the *refines* an intralevel relation. The *satisfies* relation can be used to check the consistency of a high level requirement model with respect to another model, representing lower level design information, as defined in Section 4.7.

Furthermore, the trace model offers a number of additional advantages. It offers the possibility to model traces with a richer semantics, not being restricted by a specific metamodel offering a limited set of relations. The trace metamodel offers an extension mechanism to include (or link to) existing interlevel relations, intralevel relations and within-model relations, and treat them as traces. This is possible by treating traces as being a role of other relations. The *satisfies* relation and *refines* relation are just two of these relations which can be treated as traces.

The *satisfies* relation is a bidirectional relation, used to check the consistency of both a requirement model with respect to an architectural design, and an architectural design with respect to an requirement model. We have defined a model m1 to be consistent with respect to another model, m2 if every element in m1 is satisfied by an element in m2. This is formally defined in Section 4.7. We have used this definition of consistency to allow tooling to check whether a model is kept consistent when a change occurs. We therefore check whether every element is still satisfied by a lower level design element model and the other way around.

8.3.4 Tool support

Is it possible to maintain consistency between requirements and architectural design with tool support?

We have designed tool support to ease the manual process of maintaining consistency between requirements and architectural design. To facilitate this, the tool has editors to create requirement models and architectural models. These models are created conform to the proposed metamodels. The tool allows creating trace relations in a separate editor, effectively creating an instance of the trace metamodel. This allows making a connection between a requirement model and an architectural design model (among others). Once these connections are established, these can be used to check the consistency of models. Consistency of a requirement model is checked using the *satisfy* relation of the trace model. The *satisfy* relation can furthermore be combined with the *refines* relation. This can be used to check the consistency of a requirement with respect to a transitive refinement of a satisfying architectural model.

The software developer is assisted in maintaining consistency when changes are made to one of the connected models. Altering a model element in a requirements model which is still connected to for example an architectural design model element will prompt the software developer with a message notifying the presence of this connection, and possible loss in consistency. In this way, the developer is assisted in maintaining consistency between requirements and architectural design. The tool uses a Prolog-engine to reason about the consistency of a model. This requires the tool to make an internal translation to Prolog facts and visa-versa. The architecture of the tooling is described in Section 6.3.

The tool reduces the amount of work involved in manually checking traces (which are often also manually maintained). Next to this, the tool also reduces the chance of missing certain traces which are present.

Compared to existing tooling like RequisitePro, we offer a more extendible, semantic rich model for managing requirements and architectures. The semantic information known about the requirements can be exploited for tracing purposes. When we want to trace the most expensive requirements to their stakeholders, this is possible by defining custom Prolog queries and implementing them in the tooling. This kind of extendibility is not available in existing tools. Furthermore RequisitePro only allows the tracing from a requirement document to actual code implementation, which leaves a big gap of design. Our tooling is aimed at maintaining consistency between these levels of design. In future implementation, consistency to implementation level (such as code) should also be possible. More work is however needed before this can be achieved.

8.4 Future work

From the answers on the research questions we already identified a number of possible improvements and opportunities for the future. We will discuss these opportunities briefly:

- The thesis focuses on the consistency between requirements level and architectural design level. A basis for improving the maintenance in consistency between these two levels is given. It is interesting to extend this work towards the code or implementation level. This would make possible for a requirement engineer to check whether all the requirements in a model are actually implemented in code. To achieve this, metamodels would have to be devised for the lower development levels, corresponding to the lower development phases. After these are established, also interlevel relations should be defined for connecting these lower levels. An extended version of the *satisfies* relation would have to be defined before these can be used to check certain properties of one of the connected models. Once these interlevel relations can be made, it would require furthermore the chaining of trace models, to be able to trace requirements down to the lowest level.

- Another extension is the use of additional relations in forming traces. Existing relations in both the requirements and architectural metamodel can be used in order to serve specific tracing goals. Next to these existing relations, also further investigation is possible in creating other (intralevel) trace relations. One example which is interesting to investigate is the relation between versions of models in case of system evolution. This requires thorough investigation before this can be implemented. Evolution is a very important aspect because it has close resemblance to the refinement of a model.

- An area that is touched in the discussion on architectural modelling is the need for multiple views to serve multiple stakeholders. These different views represent the same system, but describe it from a different point of view. The consistencies between these views need to be maintained. We assume that the *refine* relation can be used in order to maintain consistencies between these views.

- Further work is required in the tooling. We have implemented the basic functionality, enabling tracing to be created between model instances. Currently only the *satisfies* relation and *refines* relation can be established in the tooling. However, the use of additional relations should also be implemented. Also richer tracing should be implemented, based upon these relations. We have discussed the future extensions of the tooling in Section 6.6. The tooling should furthermore be tested thoroughly to remove any bugs and problems that currently exist.

- We have discussed the validation of both our requirements and architectural metamodel in Chapter 5. This validation is sufficient for our purposes in this thesis. In order to however use the metamodels outside the context of this thesis, a more formal validation should be performed. We have already suggested carrying out a full blown case-study.

- A final area for future improvements lies in the specification of process support for maintaining consistency. Our tooling offers a user to check the consistency of models, but does not prescribe a way how to do this. Consistency should be maintained in a systematic and process supported way. We have in this thesis focussed on the technical challenges of supporting consistency checking. Process support remains as a possibility for further research.

References

[Antón'96] A. Antón, E Liang, R Rodenstein. (1996) - A Web-Based Requirements Analysis Tool. Georgia Institute of Technology, College of Computing. In: Proceedings of the Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises, WET ICE, pp. 238-243.

[Ash'07] S. Ash. (2007) – MoSCow Prioritisation Briefing Paper. A briefing paper for senior managers outlining the principles and processes of MoSCoW Prioritisation. DSDM Consortium. http://globalfacilitatorsnetwork.org/knowledgebase/details/165/moscow-prioritisation-briefing-paper.html. Accessed 10-2007

[Bass'03] L. Bass, P. Clements, R. Kazman (2003) – Software Architecure in practice 2nd Edition. Addison Wesley. ISBN: 0-321-15495-9.

[Bass'05] L. Bass, P. Clements, R. Kazman, R. Nord (2005) – The Architecture Business Cycle Revisited: A Business Goals Taxonomy to Support Architecture Design and Analysis. Published 2005. Taken from http://www.sei.cmu.edu/news-at-sei/columns/the_architect/2005/2/architect-2005-2.htm. Accessed 10-2007.

[Belady'76] L. A. Belady, M. M. Lehman. (1976). - A model of large program development. In: IBM Systems Journal Vol. 15 No. 3: pp. 225-252.

[Bézivin'05] J. Bézivin. (2005) - On the Unification Power of Models. In: Software and Systems Modeling. Vol. 4 No. 2: pp. 171-188

[Bruin'01] H.de Bruin, H van Vliet (2001) - Feature and Feature Interaction Modeling with Feature-Solution Graphs. Vrije Universiteit, Mathematics and Computer Science Department. Amsterdam. In: Proceedings of the GCSE'01 Feature Modeling Workshop, September 9-13, pp. 1-4.

[Clevland'03] J. Cleland-Huang, C.K. Chang. (2003) – Event-Based Traceability for Managing Evolutionary Change. Published by IEEE Computer Society. In: IEEE Transactions on Software Engineering, Vol 29. No 9: pp. 796-810

[Dijkstra'82] E. W. Dijkstra. (1982) - Selected writings on computing: a personal perspective. Pg. 60–66. Springer-Verlag New York, Inc. ISBN 0–387–90652–5.

[EMF'08] Eclipse documentation (2008), http://help.eclipse.org/help33/index.jsp?topic=/org.eclipse.emf.doc//references/overview/EMF.html, accessed 30-01-08.

[Gotel'94] O.C.Z. Gotel, A.C.W. Finkelstein. (1994) – An analysis of the Requirements Traceability Problem. Imperial College of Science Technology & Medicine Department of Computing, London. In: Proceedings of International Conference on Requirements Engineering. IEEE CS Press, pp. 94-101.

[Gotel'95] O.C.Z. Gotel, A.C.W. Finkelstein. (1995) – Contribution Structures. Imperial College of Science Technology & Medicine Department of Computing, London. Second IEEE International Symposium on Requirements Engineering. In: RE IEEE Computer Society. pp 100-107

[Hoss'06] A.M. Hoss. (2006) – Ontology-based methodology for error-detection in Software-design. A Dissertation. Louisianna State University and Agricultural and Mechanical College. In: Proceedings of the 10th IASTED International Conference on Software Engineering and Applications, SEA pp. 483-488

[IEEE-610'90] The institute of Electrical and Electronics Engineers (1990) - IEEE Standard Glossary of Software Engineering Terminology. IEEE Standards Board, New York. pp 62.

[IEEE-830'98] The institute of Electrical and Electronics Engineers (1998) - IEEE Recommended Practice for Software Requirements Specifications. IEEE Standards Board, New York.

[IEEE1471'00] The institute of Electrical and Electronics Engineers (2000) - IEEE Recommended Practice for Architectural Description of Software-Intensive Systems. The institute of Electrical and Electronics Engineers (1998) - IEEE Recommended, New York.

[Jacobson'99] I. Jacobson, G. Booch, J. Rumbaugh.(1999) The Unified Software Development process, Addison-Wesley. In"IEEE Software Vol. 16 No. 3: pp 82-90

[Keet'06] C.M. Keet (2006) Part-Whole relations in Object-Role models. KRDB Research Centre, Faculty of Computer Science, Free University of Bozen-Bolzano, Italy. In: OTM Workshops 2006: Montpellier . Vol. 2: pp. 1118-1127.

[Knethen'02] A. von Knethen, B. Paech. (2002) – A survey on Tracing Approaches in Practice and Research. Fraunhofer Institut Experimentelles Software Engineering, Kaiserslautern. In: IESE-Report Nr. 095.01/E.

[Kontonya'98] G. Kotonya, I. Sommerville. (1998) - Requirements Engineering: processes and techniques. John Wiley. ISBN: 978-0-471-97208-2

[Koning'02] H. Koning, G Florijn. (2002) – Visualisatie van architecturen. In: Informatie, Ten Hagen Stam, jaargang 44.

[Kraus'07] T.U. Kraus. (2007) - Generating system documentation augmented with raceability information, using a central XML-based repository, Master Thesis.

[Kruchten'95] P.B. Kruchten. (1995) – The 4+1 View Model of Architecture. In: IEEE Software ISSN: 0740-7459. Vol. 12, No. 6: pp 42-50.

[Kurtev'03] I. Kurtev, K.G. van den Berg. (2003) – A synthesis-Based Approach to Transformations in an MDA Software Development Process. University of Twente, Enschede, The Netherlands. In: First Workshop on Model Driven Architecture: Foundations and Applications, Enschede, the Netherlands. ISSN 1381-3625. pp. 121-126.

[Lamsweerde'03] A. van Lamsweerde. (2003) – From System Goals to Software Architecture. Université catholique de Louvain, Belgium. Proceedings of the 25th International Conference on Software Engineering, pp 744-745.

[Lamsweerde'04] A. van Lamsweerde. (2004) – Goal-Oriented Requirements Engineering: A roundtrip from Research to Practice. Université catholique de Louvain, Belgium. 12th Proceedings of the IEEE International Conference on Requirements Engineering, pp. 4-7

[Lange'97] D.B. Lange, Y. Nakamura. (1997) – Object-Oriented Program Tracing and Visualisation. IEEE Computer, Vol 30, No. 55, pp 63-70.

[Lehman'06] M.M. Lehman, J. C. Fernandez-Ramil. (2006) - Software Evolution and Feedback: Theory and practice. Wiley interScience. ISBN 0470871806

[Marshall'03] F. Marshall, M. Shoenmakers. (2003) - Towards Model-Based Requirements Engineering for Web-Enabled B2B Applications. Technische Universität München, Institut für Informatik, München. Proceedings of the 10 th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems (ECBS'03). pp. 312.

[Malan'01] R. Malan, D. Bredemeyer (2001) - Defining Non-Functional Requirements. Bredemeyer Consulting, Bloomington. http://www.bredemeyer.com/pdf_files/NonFunctReq.PDF. Accessed 10-2007.

[McAffer et al'06] J. McAffer, Jean-Michel Lemieux (2006) – Eclipse Rich Client Platform, Designing, Coding and Packaging Java Applications. Addison-Wesley. ISBN: 0-321-33461-2

[Medvidovic'97] N. Medvidovic, R.N. Taylor. (1997) - A Classification and Comparison Framework for Software Architecture Description languages. Software Engineering Journal. Vol. 12 No. 1: pp. 70-93.

[Noppen'07] J.A.R. Noppen. (2007) – Imperfect information in Software Design Processes, PhD Thesis, University of Twente, 2007. ISBN: 978-90-365-2511-4. Ph.D. Thesis.

[OMG'04] Object Management Group (2004) - OMG SysML Specification. http://www.omg.org/docs/formal/07-09-01.pdf Accessed 09-2007.

[Pinheiro'03] F.A.C. Pinheiro. (2003) – Perspectives on Software requirements. Kluwer international series in Engineering and Computer Science. ISBN: 978-1-4020-7625-1

[Schätz'05] B. Schätz, A. Fleischmann, E. Geisberger, M. Pister. (2005) - Model-Based Requirement Engienering with AutoRAID. Technische Universität München, Fakultät für Informatik, München. GI Jahrestagung (2) 2005: 511-515.

[SEI'08] Software Engineering Institute. (2008) - http://www.sei.cmu.edu/str/indexes/glossary. Software Engineering Institute Glossary. Accessed 03-2008

[Sharp'99] H. Sharp, A. Finkelstein, S. Galal. (1999) - Stakeholder Identification in the Requirements Engineering Process. In: Database and Expert Systems Applications, 1999. DEXA Workshop 1999: pp. 387-391.

[SSE'08] Centre for Software Engineering. (2008) - http://www.cse.dcu.ie/essiscope/sm2/9126ref.html., ISO 9126: The Standard Reference. Accessed 13-03-2008

[Tekinerdogan'00] B. Tekinerdogan. (2000), Synthesis-Based Software Architecture Design.Print Partners Ipskamp, Enschede, ISBN 90-365-1430-4, Ph.D. Thesis. Also available through http://www.cs.bilkent.edu.tr/~bedir/PhDThesis/index.htm.

[Vicente'07] C. Vicente-Chicote, B. Moros, A. Toval. (2007)- REMM-Studio: an Integrated Model-Driven Environment for Requirements Specification, Validation and Formatting. In Journal of Object Technology Vol. 6 No 9: pp. 437-454

[Volere'07] Volere Requirements Resources (2007) - http://www.volere.co.uk. Requirements Specification Template. Accessed 06-2007.

[Wieringa'03] R. Wieringa (2003) – Design methods for reactive systems, Yourdon Statemate and the UML. University of Twente, The Netherlands. Published by Morgan Kaufmann Publishers, San Francisco. ISBN 1-55860-755-2.

[Wieringa'95] R. Wieringa (1995) – An introduction to requirements traceability. Technical report IR-389, Faculty of Mathematics and Computer Science, Vrije Universiteit, Amsterdam. Esprit Project 2RARE.

[Zelkowitz'97] M. V. Zelkowitz, D. R. Wallace (1997) - Experimental validation in software engineering. In Information and Software Technology. Vol. 39 No 11: pp. 735-743.

[Zelkowitz'98] M. V. Zelkowitz, D. R. Wallace (1998) - Experimental Models for Validating Technology. In IEEE Computer Vol. 31, No 5: pp: 23 - 31

[Zelkowitz'07] M.V. Zelkowitz (2007) - Techniques for Empirical Validation. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) 4336 LNCS, pp. 4-9. Springer-Verlag Berlin Heidelberg 2007

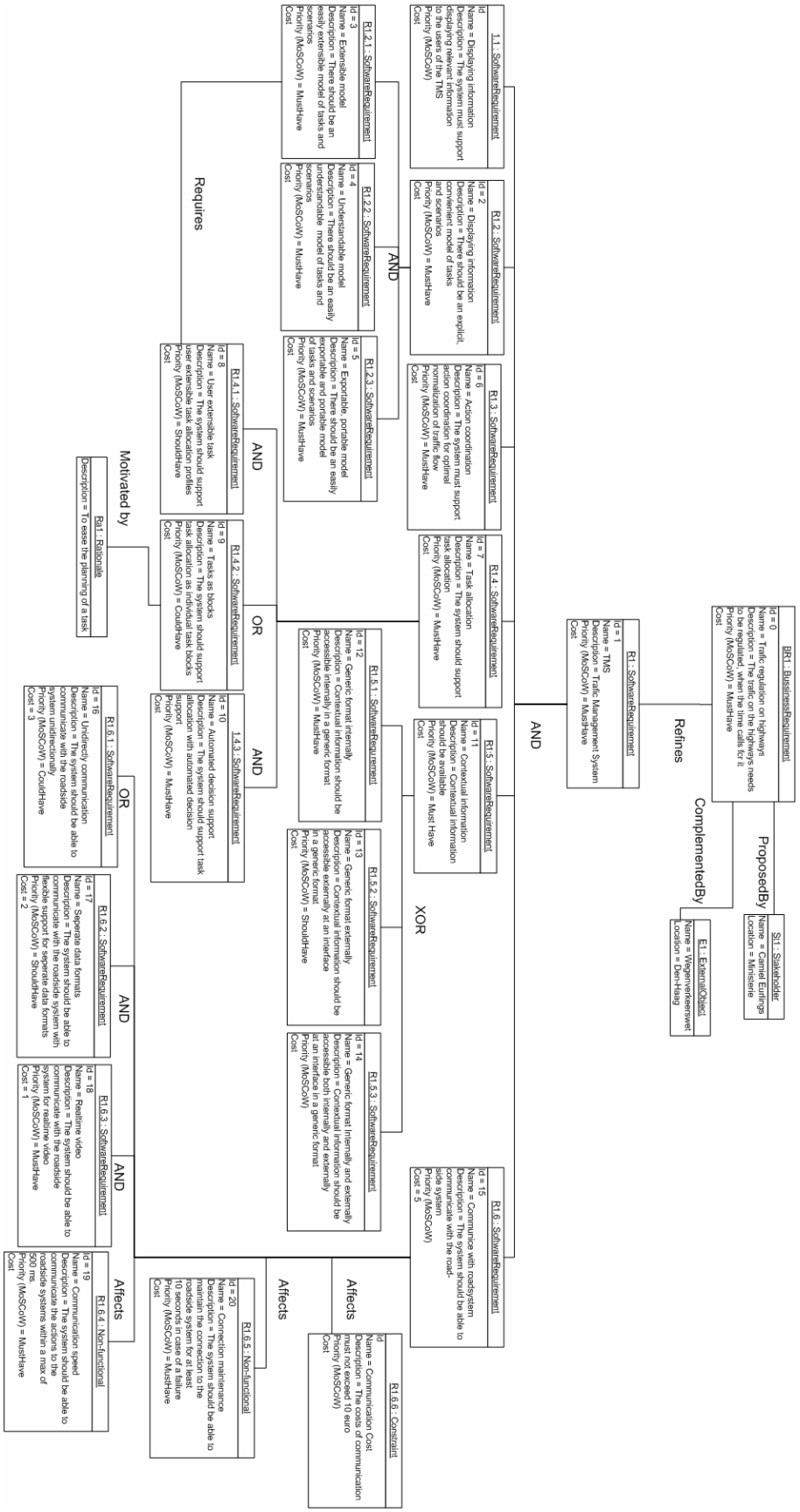
Appendix A. Requirements TMS

The original requirements for the task allocation part of the Traffic Management System, as stated in [Noppen'07]. The task allocation part of the TMS handles allocation based on scenarios and available traffic information.

Functional Requirement	
1	The TMS must support displaying relevant information to the users of the TMS
2	There should be an explicit, convenient model of tasks and scenarios
2.1	There should be an easily extensible model of tasks and scenarios
2.2	There should be an easily understandable model of tasks and scenarios
2.3	There should be an easily exportable and portable model of tasks and sce-narios
3	The system must support action coordination for optimal normalization of traffic flow
4	The system should support task allocation
4.1	The system should support user extensible task allocation profiles
4.2	The system should support task allocation as individual task blocks
4.3	The system should support task allocation with automated decision support
5	Contextual Information should be accessible
5.1	Contextual Information should be accessible internally in a generic format
5.2	Contextual Information should be accessible externally at an interface in a generic format
5.3	Contextual Information should be accessible both internally and externally at an inter-face in a generic format
6	The TMS should be able to communicate with the roadside system
6.1	The Traffic Management System should be able to communicate with the roadside system unidirectional
6.2	The Traffic Management System should be able to communicate with the roadside system with flexible support for separate data formats
6.3	The Traffic Management System should be able to communicate with the roadside system for real-time video

Non-Functional Requirement & Constraints	
1	The system should be able to communicate the actions to the roadside systems within a max of 500 ms.
2	The system should be able to maintain the connection to the roadside system for at least 10 seconds in case of a failure
3	The costs of communication must not exceed 10k euro

The same requirement of the Traffic Management System, represented as an instance of our requirements metamodel is presented below.



Appendix B. Requirements Forest project

The original requirements on the proposed solutions of the Forest project. These requirements where used to create the tooling for the Forest project [Kraus'07].

1. Central repository: All information is to be stored in a repository, that is integrated with the location of the project's source files. This way, it is possible to verify references to implementation artefacts in the file system. Furthermore, the repository acts as a frame holding all information together. This should improve the coupling and consistency between documents.
2. Store all types of information during the system's entire lifecycle: Some existing tools only focus on one phase of the lifecycle, e.g. requirements management or testing. We aim at storing all types of information, to be able to trace requirements in a complete way and to improve consistency
3. Automated consistency checking and requirements traceability: A tool should be available that can perform a number of consistency checks automatically. Checks that should at least be available, are:
   3.1. Each requirement should be implemented somewhere,
   3.2. All implementation artefacts need a least one requirement to explain their presence,
   3.3. For external references to the file system, the actual existence of the files should be verified.
4. Support for reuse: Artefacts that might be reused, like a set of requirements or a (sub)system designed for common problems, should be defined abstractly in reusable systems. Reuse now simply becomes a matter of extending these systems, and it should be verified automatically that all local and all inherited requirements are implemented. Furthermore, a concrete system description should be available for every inherited abstract system definition. Sets of domain-specific requirements, standards, or patterns can now be defined once and reused many times.
5. Generation of dynamic, personalized documentation: A user should be able to generate documentation specific to his role. For example, a requirements engineer will typically be interested in generating requirements documentation. Furthermore, a user should be able to select the part of the system of which documentation should be generated, as well as the desired level of abstraction. An architect is typically interested in the design of an entire system, at a high level of abstraction; whereas a developer is more interested in a small part of the system, but including all the details. This solves the problem of having static documents providing only a single view on the information.
6. Document any system: Many systems are limited to documenting software systems only. Since this can be a limitation when creating e.g. embedded systems, the proposed method should allow to describe anything that can be expressed as a "system", not only software systems. A commonly used example of a "system" to describe is a business hierarchy (i.e. a company with a board of directors, several business led by a manager, employees in each business unit, etc).
7. Easy to use: The user should be able to fill in as much or as little information as he likes. There should be no blocking error messages or dialog screens that enforce the user to enter more information or invest more time at a certain moment.
8. Low threshold to start using Forest: It should be very little work to define the structure of a system in terms of subsystems. Combined with the previous requirement, the user should be free to explore the possibilities and use as much or as little of the available possibilities as desired.
9. Integrated in development environment: We expect an advantage in efficiency when documentation is well-integrated in the development environment. A developer should be able to find the desired documentation, or the actual implementation of a requirement, faster than in the current situation, by means of additional navigation between implementation and documentation.
10. Separate tools and content: By storing the information artefacts and the relations between them in an open and readable format, e.g. an XML format, a Forest repository is independent of specific tools for viewing or modifications. "Anyone" should be able to build a viewer or editor on top of such an XML-based repository.
11. Open and extendable: Anyone should be able to build tools and extensions for a Forest repository, and the format should still be readable in the future.
12. Not intended for code generation: The Forest project aims at documenting systems, which is in itself already quite a complex task. In order to generate code for a complex system, that system must first be fully and properly specified. Today, this is still a difficulty so this is our primary goal. Perhaps code generation becomes of interest in a later phase of the Forest project.

Appendix C. Requirements Tracing Toolkit

Requirements for our tooling, derived from the problem statement. These requirements where not officially formalized at the beginning of the project, but where implied by the project statement.

1. The tooling shall be able to integrate into the Eclipse environment
2. The system shall enable the user to enter requirement information into the system.
   - 2.1. There shall be easy to use editors to enter the requirement information.
   - 2.2. The requirement information shall be stored into models.
   - 2.3. The modelling of requirements shall be conform to a defined requirement metamodel
   - 2.4. The models shall each be stored in a separate repository to enable extending the editors
3. The system shall enable the user to enter (architectural) design information into the system.
   - 3.1. There shall be easy to use editors to enable the storing of the design.
   - 3.2. The design information shall be stored into models.
   - 3.3. The modelling of design shall be conform to a defined design metamodel
   - 3.4. The models shall each be stored in a separate repository to enable extending the editors
4. The system shall enable the user to create (trace) relations between model elements
   - 4.1. The system shall able the user to create the following relations
     - 4.1.1.Between different metamodel instances
     - 4.1.2.Between the same metamodel instances.
   - 4.2. These relations shall be stored into a separate model for further analysis
5. The system shall keep a repository for every model instance
   - 5.1. The information shall be stored in a open format
   - 5.2. It shall be possible to create different editors on the repository
6. The system shall enable the following trace analysis to be performed:
   - 6.1. Implementation of requirements into design
   - 6.2. Presence of a requirement for every design element
   - 6.3. Refinement of one model (element) into another model (element).
   - 6.4. It shall be possible to extend the analysis for the purpose of tracing
     - 6.4.1.Also existing relations shall possibly be used for tracing
   - 6.5. The reasoning about the tracing shall be carried out using the Prolog engine
7. The tracing engine in the system shall upon deletion of requirement or design element, automatically inspect the possible consistency implications between requirements and design through the tracing mechanism

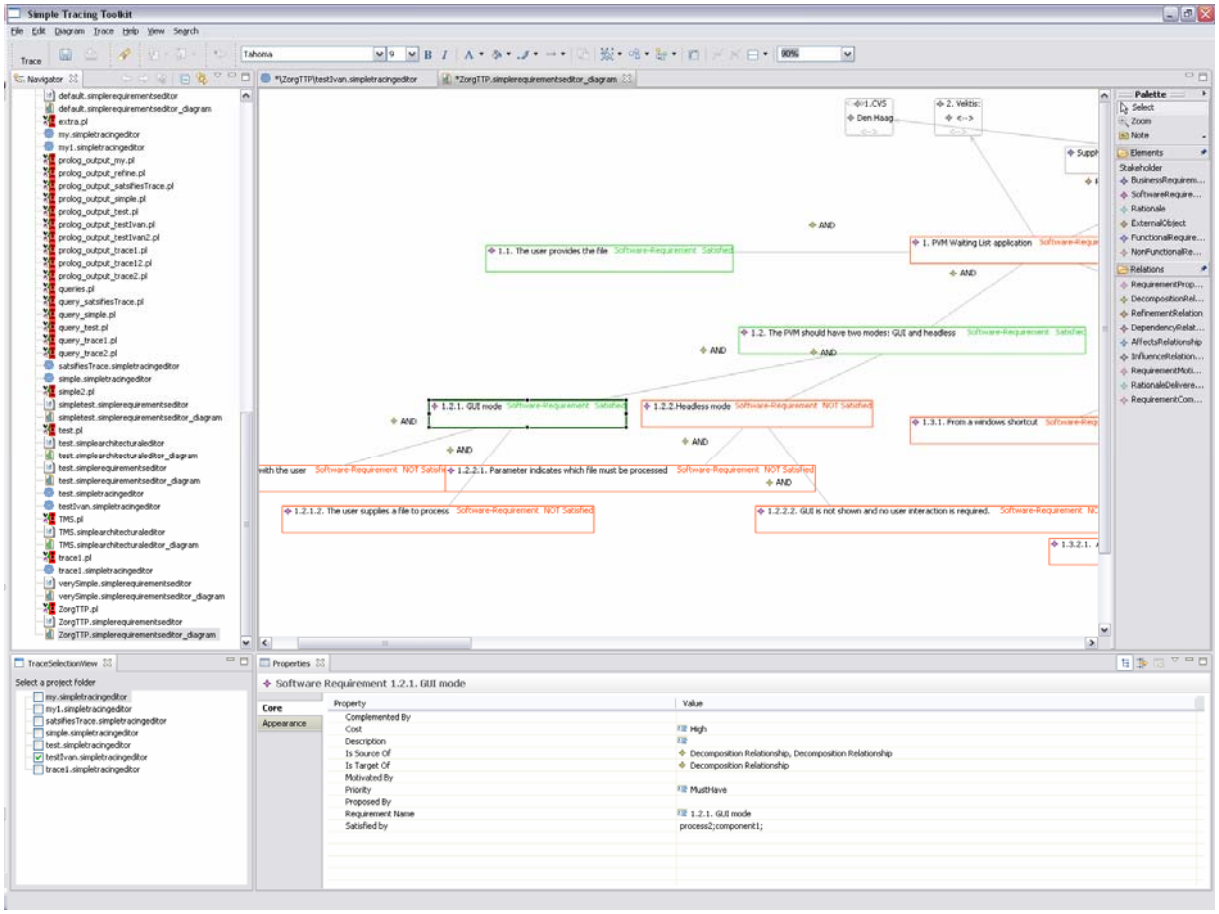
Appendix D. Comparison criteria for comparing Forest Tool with the Tracing Toolkit

The full framework of comparison criteria, grouped per source of criteria. These are: the requirements of the Forest project, the requirements of the Tracing Toolkit, general criteria for tooling and specific criteria for tracing.

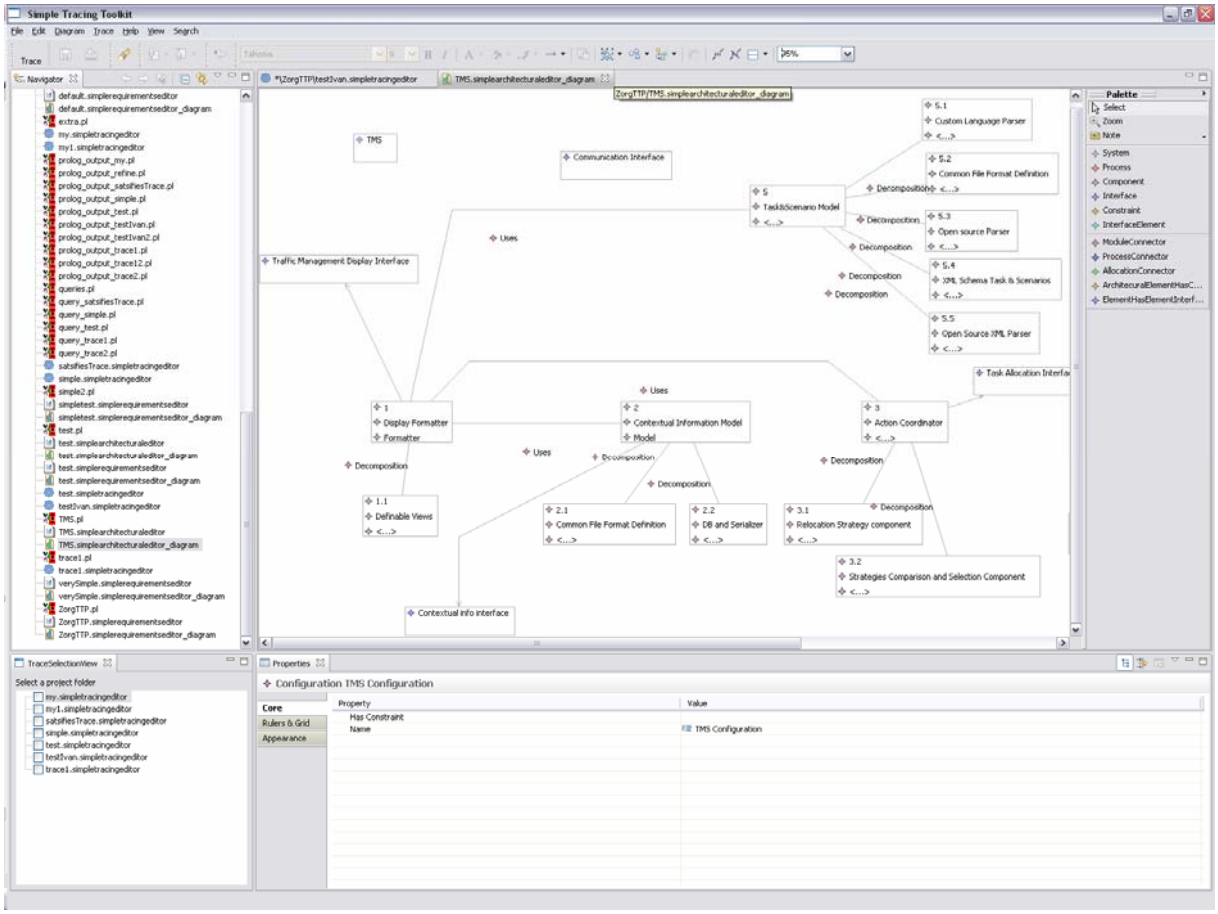
- Criteria based upon the requirements of the Forest project:
  - Storage location
  - Kind of information stored
  - Openness of the storage format.
  - Extendibility of the storage format.
  - Kind of (automated) traceability checks:
    - Implementation of requirements
    - Presence of requirements for design artefacts
    - Presence of externally referred files
  - Reuse of artefacts
  - Generation of documentation
  - Easy of use through non-mandatory information. (Use as see fit)
  - Defining of (sub) systems should be easy
  - Integration with the Development environment
- Criteria based upon the requirements of the Tracing Toolkit (the project in this thesis):
  - Integration with the Development environment
  - Manner of Storage
  - Openness of the storage format.
  - Extendibility of the storage format for different tooling
  - Creation of traces
  - Storage of the traces.
  - Possible analyses of traceability
  - Extendibility of analyses
  - Way of performing trace analysis
  - Automated support for maintaining consistency
- Criteria based upon general criteria for tooling:
  - Usability
    - Creation of requirements
    - Creation of design
  - Extendibility
    - Support for different traces to be created (Prolog)
    - Support for different editors
  - Scalability
    - Requirements
    - Design – subsystems in Forest
- Criteria specifically based upon tracing
  - Representation / Creation of traces: trace matrix is generated by Forest.
  - Specification of trace relations
    - Change management: what parts are affected by a change. (Impact analysis, not done)
    - Derivation analysis: tracing from low level to upper level
    - Coverage analysis: Which requirements are implemented.
    - Rich traceability: Store design decisions (rationale)
    - Automatic discovery of links is desirable, but not manually.
  - Level of detail in modelling
    - Design – subsystems in Forest
  - Freedom in Relations
    - Limited in Forest

Appendix E. Screenshot of the Tracing Toolkit

The first screenshot shows the requirement editor, opening the ZorgTTP requirement document, that we used to validate the requirements metamodel.



The second screenshot shows the (architectural) design editor, opening the TMS design document, used to validate the architectural metamodel.



Appendix F. Example Model Serialisation

An example XML- serialisation of the requirement model show in Figure 27.

```
<?xml version="1.0" encoding="UTF-8"?>
<sre:RequirementModel xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:sre="sre"
UID="r_tracetoolkit_toolkit1_RequirementModel_0">
  <ContainsRelationship xsi:type="sre:DecompositionRelationship"
SourceRequirement="r_tracetoolkit_toolkit1_SoftwareRequirement_0"
TargetRequirement="r_tracetoolkit_toolkit1_SoftwareRequirement_1"
UID="r_tracetoolkit_toolkit1_DecompositionRelationship_0"/>
  <ContainsRelationship xsi:type="sre:DecompositionRelationship"
SourceRequirement="r_tracetoolkit_toolkit1_SoftwareRequirement_0"
TargetRequirement="r_tracetoolkit_toolkit1_SoftwareRequirement_2"
UID="r_tracetoolkit_toolkit1_DecompositionRelationship_1"/>
  <ContainsRequirement xsi:type="sre:SoftwareRequirement"
RequirementName="Provide editors for all Meta-models"
IsSourceOf="r_tracetoolkit_toolkit1_DecompositionRelationship_0
r_tracetoolkit_toolkit1_DecompositionRelationship_1"
UID="r_tracetoolkit_toolkit1_SoftwareRequirement_0"/>
  <ContainsRequirement xsi:type="sre:SoftwareRequirement"
RequirementName="Provide an Requirement-editor"
IsTargetOf="r_tracetoolkit_toolkit1_DecompositionRelationship_0"
UID="r_tracetoolkit_toolkit1_SoftwareRequirement_1"/>
  <ContainsRequirement xsi:type="sre:SoftwareRequirement"
RequirementName="Provide an Design-editor"
IsTargetOf="r_tracetoolkit_toolkit1_DecompositionRelationship_1"
UID="r_tracetoolkit_toolkit1_SoftwareRequirement_2"/>
</sre:RequirementModel>
```

Appendix G. Example Prolog Serialisation

An example Prolog serialisation of the requirement model show in Figure 27.

```
attr_requirementname('r_tracetoolkit_toolkit1_softwarerequirement_0','provide editors for all
meta-models').
attr_requirementname('r_tracetoolkit_toolkit1_softwarerequirement_1','provide an requirement-
editor').
attr_requirementname('r_tracetoolkit_toolkit1_softwarerequirement_2','provide an design-
editor').
attr_uid('r_tracetoolkit_toolkit1_requirementmodel_0','r_tracetoolkit_toolkit1_requirementmode
l_0').
attr_uid('r_tracetoolkit_toolkit1_decompositionrelationship_0','r_tracetoolkit_toolkit1_decomp
ositionrelationship_0').
attr_uid('r_tracetoolkit_toolkit1_decompositionrelationship_1','r_tracetoolkit_toolkit1_decomp
ositionrelationship_1').
attr_uid('r_tracetoolkit_toolkit1_softwarerequirement_0','r_tracetoolkit_toolkit1_softwarerequ
irement_0').
attr_uid('r_tracetoolkit_toolkit1_softwarerequirement_1','r_tracetoolkit_toolkit1_softwarerequ
irement_1').
attr_uid('r_tracetoolkit_toolkit1_softwarerequirement_2','r_tracetoolkit_toolkit1_softwarerequ
irement_2').
class_decompositionrelationship('r_tracetoolkit_toolkit1_decompositionrelationship_0').
class_decompositionrelationship('r_tracetoolkit_toolkit1_decompositionrelationship_1').
class_requirementmodel('r_tracetoolkit_toolkit1_requirementmodel_0').
class_softwarerequirement('r_tracetoolkit_toolkit1_softwarerequirement_0').
class_softwarerequirement('r_tracetoolkit_toolkit1_softwarerequirement_1').
class_softwarerequirement('r_tracetoolkit_toolkit1_softwarerequirement_2').
ref_containsrelationship('r_tracetoolkit_toolkit1_requirementmodel_0','r_tracetoolkit_toolkit1
_decompositionrelationship_0').
ref_containsrelationship('r_tracetoolkit_toolkit1_requirementmodel_0','r_tracetoolkit_toolkit1
_decompositionrelationship_1').
ref_containsrequirement('r_tracetoolkit_toolkit1_requirementmodel_0','r_tracetoolkit_toolkit1_
softwarerequirement_0').
ref_containsrequirement('r_tracetoolkit_toolkit1_requirementmodel_0','r_tracetoolkit_toolkit1_
softwarerequirement_1').
ref_containsrequirement('r_tracetoolkit_toolkit1_requirementmodel_0','r_tracetoolkit_toolkit1_
softwarerequirement_2').
ref_issourceof('r_tracetoolkit_toolkit1_softwarerequirement_0','r_tracetoolkit_toolkit1_decomp
ositionrelationship_0').
ref_issourceof('r_tracetoolkit_toolkit1_softwarerequirement_0','r_tracetoolkit_toolkit1_decomp
ositionrelationship_1').
ref_istargetof('r_tracetoolkit_toolkit1_softwarerequirement_1','r_tracetoolkit_toolkit1_decomp
ositionrelationship_0').
ref_istargetof('r_tracetoolkit_toolkit1_softwarerequirement_2','r_tracetoolkit_toolkit1_decomp
ositionrelationship_1').
ref_sourcerequirement('r_tracetoolkit_toolkit1_decompositionrelationship_0','r_tracetoolkit_to
olkit1_softwarerequirement_0').
ref_sourcerequirement('r_tracetoolkit_toolkit1_decompositionrelationship_1','r_tracetoolkit_to
olkit1_softwarerequirement_0').
ref_targetrequirement('r_tracetoolkit_toolkit1_decompositionrelationship_0','r_tracetoolkit_to
olkit1_softwarerequirement_1').
ref_targetrequirement('r_tracetoolkit_toolkit1_decompositionrelationship_1','r_tracetoolkit_to
olkit1_softwarerequirement_2').
```

Appendix H. Example Prolog query

A Prolog query checking the implementation of a requirement, using transitive refinement.

```
requirementImplementation(R, AM, L):-satisfies_trace(RM, AM), satisfies(R, RM, L).
requirementImplementation(R, AM, L):-refines_trace(AM1, AM), requirementImplementation(R, AM1,
L1), iterateComponents(R, L1, L, AM1).

iterateComponents(R, [], [], _).
iterateComponents(R, [X|Y], L, AM):-refines(X, L1, AM), iterateComponents(R, Y, L2, AM),
append(L1, L2, L).

satisfies(R, RM, L) :- ref_containsrequirement(RM, R), class_softwarerequirement(R),
satisfiesAll(R, L).

satisfiesAll(X, L) :- findall(Y, satisfiesSingle(X, Y), L).

satisfiesSingle(X,Y) :-

       ref_tracesource(SATREL,SA),
       ref_tracetarget(SATREL,TA),
       class_sourceartifact(SA),     %SA = Source Artefact
       class_targetartifact(TA),     %TA = Target Artefact
       attr_artifact(SA,X),
       attr_artifact(TA,Y),
       ref_consistsof(TR,SATREL),
       class_satisfies(SATREL),      %SATREL = Satisfies relation
       class_trace(TR),                      %TR = Trace
       attr_sourcemodel(TR,SM),      %SM = SourceModel
       attr_targetmodel(TR,TM),      %TM = TargetModel
       ref_containstraces(TRM,TR),
       class_tracemodel(TRM).        %TRM = TraceModel

refines(AC, L, AM) :- ref_containselement(AM, AC), class_component(AC), refinesAll(AC, L).

refinesAll(X, L) :- findall(Y, refinesSingle(X, Y), L).

refinesSingle(X,Y) :-

       ref_tracesource(REFREL,SA),
       ref_tracetarget(REFREL,TA),
       class_sourceartifact(SA),     %SA = Source Artefact
       class_targetartifact(TA),     %TA = Target Artefact
       attr_artifact(SA,X),
       attr_artifact(TA,Y),
       ref_consistsof(TR,REFREL),
       class_refines(REFREL),%REFREL = refinement relation
       class_trace(TR),                      %TR = Trace
       attr_sourcemodel(TR,SM),      %SM = SourceModel
       attr_targetmodel(TR,TM),      %TM = TargetModel
       ref_containstraces(TRM,TR),
       class_tracemodel(TRM).        %TRM = TraceModel


componentOrigin(AC, LR) :- class_component(AC), ref_containselement(AM, AC), findall(R,
relatedRequirement(R, AC, AM), LR).

relatedRequirement(R, AC, AM) :-  requirementImplementation(R, AM, L), member(AC, L).

subset([], _).
subset([X | Y], Z):-member(X, Z), subset(Y, Z).

member(X, [X|_]).
member(X,[_|L]):-member(X,L).
```