

MASTER'S THESIS

---

# Tool Support for Change Impact Analysis in Requirement Models

Exploiting semantics of requirement relations as traceability  
relations

---

*Author:*

Wietze Spijkerman

*Committee:*

dr. I.Kurtev MSc. (1<sup>st</sup> supervisor)

A.Göknil MSc.

dr. M. Daneva

*Research Group:*

Software Engineering

Faculty of Electrical  
Engineering, Mathematics and  
Computer Science

University of Twente

October 26, 2010



# Abstract

In the process of software engineering, eliciting requirements yields the first software artifacts; software requirements specifications. Software requirements are used to describe what a software system should do and are used to validate if the implementation satisfies the needs determined for it. From the moment that the first artifacts are produced, they are subject to change.

Software artifacts are related to each other through traceability information. Using the traceability information, change impact analysis is performed and it is determined which related software artifacts require changes as well.

However, without additional semantics of traceability information an explosion of impacts occurs. This means that change impact analysis yields all related artifacts and causes many false positives. Existing literature states that additional semantics should be employed to counter the explosion of impacts.

Recent research yielded a metamodeling approach for requirements models that describes semantics for requirements relations. Tool support has been developed following this approach, that allows for inferencing and consistency checking of requirements models using these relations. However, the tool provides no support for performing change impact analysis.

In this work, the semantics of requirements relations are exploited for the use of change impact analysis. This is done by identifying a classification of change for the formalized requirements model. Consequently different rationales of change are identified. It is determined that domain changes drive the change impact analysis. By combining the formalization for the rationale of change and the determined change classification, the semantics of requirements relations allow for a more precise propagation of change. It also enables semi-automatic support for relation validation, as well as identifying inconsistencies of multiple changes.

Existing tool support for the metamodeling approach is extended to support change impact analysis using the semantics of requirements relations. The approach is then evaluated by performing an example case study. The example case study yields that using semantics of requirements relation for change impact analysis requires additional effort. The investment of additional effort yields a more precise change impact analysis and reduces the problem of the explosion of impacts.



# Acknowledgements

The author would like to thank his parents and Floor van Doorn for their unrelinquished support during his journey through academic education. Without their support the journey would undoubtedly have been a longer one.

The author would like express his gratitude to his original supervisors Klaas van den Berg, Arda Göknül and Ivan Kurtev for their guidance and supervision during the work that led to the writing of this thesis.

Lastly, the author would like to thank Berteun Damman for his willingness to share his extensive knowledge on L<sup>A</sup>T<sub>E</sub>X.

*Wietze Spijkerman, October 2010, Deventer*

*‘We are all shaped by the tools we use, in particular: the formalisms we use shape our thinking habits, for better or for worse, and that means that we have to be very careful in the choice of what we learn and teach, for unlearning is not really possible.’*

- Edsger Wybe Dijkstra (May 11, 1930 – August 6, 2002)



# Contents

|          |  |           |
|----------|--|-----------|
| <b>1</b> | <b>Introduction</b>                              | <b>1</b>  |
| 1.1      | Introduction . . . . .                           | 1         |
| 1.2      | Problem statement . . . . .                      | 1         |
| 1.3      | Approach . . . . .                               | 3         |
| 1.4      | Contributions . . . . .                          | 3         |
| 1.5      | Outline of the document . . . . .                | 4         |
| <b>2</b> | <b>Concepts and background</b>                   | <b>7</b>  |
| 2.1      | Introduction . . . . .                           | 7         |
| 2.2      | Model Driven Engineering . . . . .               | 7         |
| 2.3      | Requirements Engineering . . . . .               | 8         |
| 2.4      | Change Impact Analysis . . . . .                 | 11        |
| 2.5      | Conclusion . . . . .                             | 13        |
| <b>3</b> | <b>Formalized requirements &amp; relations</b>   | <b>15</b> |
| 3.1      | Introduction . . . . .                           | 15        |
| 3.2      | Requirements metamodel . . . . .                 | 15        |
| 3.3      | Formalization of requirements . . . . .          | 16        |
| 3.4      | Formalization of requirement relations . . . . . | 17        |
| 3.5      | Conclusion . . . . .                             | 18        |
| <b>4</b> | <b>Change classification</b>                     | <b>19</b> |
| 4.1      | Introduction . . . . .                           | 19        |
| 4.2      | Structure of requirements . . . . .              | 19        |
| 4.3      | Semantics of requirement changes . . . . .       | 23        |
| 4.4      | Semantics of change rationale . . . . .          | 26        |
| 4.5      | Conclusions . . . . .                            | 29        |
| <b>5</b> | <b>Propagation &amp; consistency checking</b>    | <b>31</b> |
| 5.1      | Introduction . . . . .                           | 31        |
| 5.2      | Change propagation . . . . .                     | 31        |
| 5.3      | Change consistency checking . . . . .            | 45        |
| 5.4      | Discussion of the approach . . . . .             | 47        |
| 5.5      | Conclusion . . . . .                             | 49        |

|          |  |           |
|----------|--|-----------|
| <b>6</b> | <b>Tool support</b>                            | <b>51</b> |
| 6.1      | Introduction . . . . .                         | 51        |
| 6.2      | Requirements for TRIC-CIA . . . . .            | 51        |
| 6.3      | Architecture of TRIC . . . . .                 | 55        |
| 6.4      | Design . . . . .                               | 57        |
| 6.5      | Implementation . . . . .                       | 62        |
| 6.6      | Tool usage . . . . .                           | 65        |
| 6.7      | Conclusion . . . . .                           | 72        |
| <b>7</b> | <b>Evaluation</b>                              | <b>73</b> |
| 7.1      | Introduction . . . . .                         | 73        |
| 7.2      | Example case study . . . . .                   | 73        |
| 7.3      | Comparison of approaches . . . . .             | 77        |
| 7.4      | Conclusion . . . . .                           | 83        |
| <b>8</b> | <b>Conclusion</b>                              | <b>85</b> |
| 8.1      | Introduction . . . . .                         | 85        |
| 8.2      | Summary . . . . .                              | 85        |
| 8.3      | Answers to the research questions . . . . .    | 86        |
| 8.4      | Future work . . . . .                          | 88        |
| <b>A</b> | <b>CMS Requirements Specification Document</b> | <b>95</b> |
| <b>B</b> | <b>Formalization of CMS requirements</b>       | <b>99</b> |



# Nomenclature

|          |  |
|----------|--|
| AIS      | Actual Impact Set  |
| CIA      | Change Impact Analysis   |
| CIP      | Change Impact Prediction   |
| CIS      | Candidate Impact Set   |
| CMS      | Course Management System   |
| CNF      | Conjunctive Normal Form  |
| DAO      | Data Access Object   |
| EMOF     | Essential Meta Object Facility                                   |
| FOL      | First Order Logic  |
| MDE      | Model Driven Engineering   |
| MOF      | Meta Object Facility   |
| OWL      | Web Ontology Language  |
| QuadREAD | Quality-Driven Requirements Engineering and Architectural Design |
| RCP      | Rich Client Platform (of Eclipse framework)                      |
| RE       | Requirements Engineering   |
| SE       | Software Engineering   |
| SHA      | Secure Hashing Algorithm   |
| SIS      | Starting Impact Set  |
| SWEBOK   | Software Engineering Body of Knowledge                           |
| TRIC     | Tool for Requirement Inferencing and Consistency checking        |
| UI       | User Interface   |
| UML      | Unified Modeling Language  |
| XML      | Extensible Modeling Language                                     |



# Chapter 1

## Introduction

### 1.1 Introduction

In software engineering, one of the first steps towards realizing the software product is eliciting requirements. These requirements are made more concrete by the architectural design of the software. From the start of the software development process, until the end of the system's lifecycle, the environment in which the system is used evolves. New requirements are introduced or existing ones are changed. For the software system to remain competitive in the environment in which it is used, these changed requirements should be reflected by changes in the architectural design, which are then in turn implemented by changes in the detailed design and so on. An overview of artifacts in the software development process and the evolution of these artifacts is depicted in Figure 1.1.

Overlooking required changes in early phases of the software development process leads to increased costs when these required changes are detected during later phases. The Quality-Driven Requirements Engineering and Architectural Design (QuadREAD) project aims at bridging the gap between early analysis phases and subsequent realization phases.

By performing change analysis on the evolved requirements, the impact of the changing requirement on other requirements is determined. This analysis can be regarded as getting the correct representation of what the system should do. The sum of changed requirements can then be traced to their related architectural design artifacts to determine required changes in the architecture. These artifacts describe how the system should be made. The work in this thesis focuses on performing analysis of evolving requirements and their resulting impacts on related requirements.

### 1.2 Problem statement

Change impact analysis (CIA) is often performed using traceability information. Traceability information consists of trace links that relate artifacts from same or different types to each other. By following the traceability links from the changed artifact a reachability analysis is performed. This analysis yields the prediction of possibly impacted artifacts, which possibly require changing.

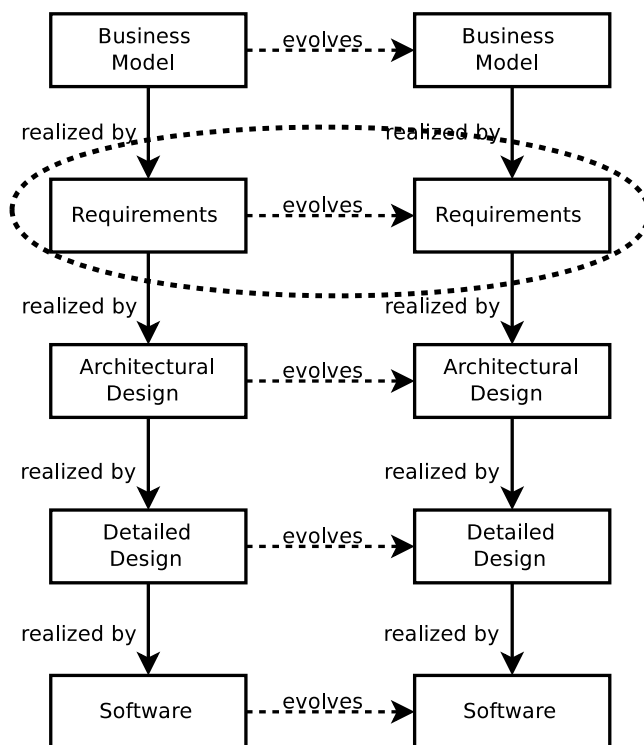


Figure 1.1: Overview of software engineering process. The dotted outline indicates the scope of this thesis.

Without the use of semantics, all reachable artifacts following traceability links will be yielded by the change impact prediction. Bohner et al. formulate this problem as the ‘explosion of impacts’, which yields imprecise results, e.g. a lot of false positives and negatives[1]. Hence, the requirement engineer may have to analyze all predicted impacts resulting from a single change. Bohner et al. state that ‘change impact analysis must employ additional semantic information to increase the accuracy of finding more valid impacts’[1]. However, current state of the art tool support does not employ these additional semantics.

This leads to the following problem statement:

*There is a gap between current and desired change impact analysis tool support. Current tool support lacks well-defined semantics for performing change impact analysis for changing artifacts. Desired tool support uses semantics for performing precise change impact analysis.*

This problem statement leads to the following research questions::

- R1 *Can CIA using semantics of traceability information in requirements models be provided with tool support?*
- R2 *What is the change classification for requirements of formalized requirements model?*

- R3 *What are the propagation results using this classification of change?*
- R4 *How does CIA using semantics of traceability information compare to CIA without semantics?*

In this work CIA is performed in the context of requirements models. Göknil et al.[2] propose a metamodeling approach to represent requirements and relations between requirements that has well-defined semantics. Tool support is available for constructing requirements models following this approach, but support for performing CIA using semantics of trace relations is lacking.

### 1.3 Approach

In this work the core requirements metamodel of Göknil et al.[2] is used for the semantics of requirements and requirements relations. The composition of textual requirements using the requirement primitives as described by Wasson[3] is investigated to determine what kind of changes can be made to textual requirements. These are then mapped to the formalized requirements. Using the mapping onto the formalized requirements, the classification of change for the formalized requirements is determined.

The different sources that cause changing requirements are investigated. By identifying the differences between the different rationales of change, the semantics of rationale of change for requirements is determined. Using the formalization of evolving requirements, an exhaustive case analysis is performed for each combination of change type and requirements relation. This analysis yields change propagation alternatives for each case. Using semantics of change propagation, relation validation and impact consistency checking can be performed.

By exploring the change propagation alternatives over possible propagation paths, change impact prediction using the determined semantics can be performed. This is realized by a tool that extends the current tool ‘Tool for Requirements Inferencing and Consistency checking’ (TRIC) with support for CIA. The approach of exploiting semantics of requirements relations as trace relations is evaluated by an example case study both with and without semantics.

### 1.4 Contributions

This thesis provides the following contributions:

*A classification of change for formalized requirements.*

In Chapter 4 the structure of textual requirements is mapped to the formalized requirements. From the resulting mapping, classification of change is identified for the formalized requirement.

*A formalization for the rationale of change.*

In Chapter 4 formalization for the rationale of change is given, to define domain changes as the needed propagation of change that causes the requirements model to change.

*Change impact alternatives for the classification of change and formalized requirements relation types.*

In Chapter 5 an exhaustive case analysis is performed to derive change impact alternatives for each change type and the formalized requirements relations.

*Tool support for CIA using semantics of requirements relations.*

In Chapter 6 the implementation is described, which is in turn evaluated in Chapter 7 by using an example case study.

## 1.5 Outline of the document

The outline of this thesis is depicted in Figure 1.2.

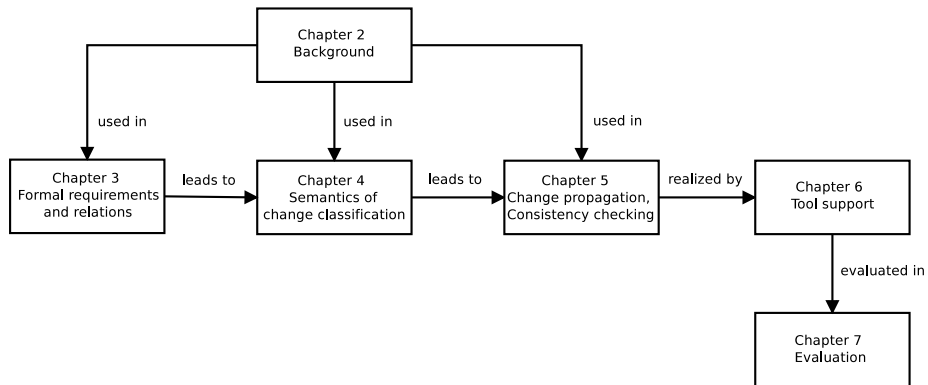


Figure 1.2: Outline of the thesis, excluding Chapter 1 and 8

**Chapter 2** introduces basic concepts used in this thesis, essential for understanding the work.

**Chapter 3** describes the formalized requirements and formalized requirements relations as described by Göknil et al.[2].

**Chapter 4** describes the mapping of the structure of textual requirements to the formalized requirement, that lead to the classification of change for the formalized requirements model. The classification of change is then formalized. Classification of change rationale in literature is described, and a formalization for refactoring and domain changes is given.

**Chapter 5** describes the rules for the propagation of change. By performing a case analysis for each change type and requirements relation, impact alternatives are determined. For each of these propagation types, rules are described for inconsistency checking in the case of requirements with multiple impacts.

**Chapter 6** describes the tool support. High level requirements are given for the change impact analysis support for the tool. The adapted architecture is described, together with the design and implementation. The chapter elaborates on use of the main features of the tool.

**Chapter 7** describes comparison of change impact analysis performed with and without using semantics of requirement relations.

**Chapter 8** describes the conclusion of the thesis and future work.





# Chapter 2

## Concepts and background

### 2.1 Introduction

This chapter introduces the basic concepts used in this thesis. In literature concepts can have multiple definitions and here the definitions used in this work are described. In section 2.2 the Model Driven Engineering software development paradigm is described. Section 2.3 describes Requirements Engineering (RE), software requirements and traceability. In section 2.4 change impact analysis is described together with the problem of ‘explosion of impacts’. This chapter is summarized in section 2.5.

### 2.2 Model Driven Engineering

Model Driven Engineering (MDE) is a software development paradigm. MDE gives basic principles for the use of models as primary engineering artifacts in software development. Models abstract from reality and help understanding, communication and analysis. A model is defined by Bézivin and Gerbé[4] as:

*‘A model is a simplification of a system built with an intended goal in mind. The model should be able to answer questions in place of the actual system’*

In the context of MDE, some authors such as Kleppe et al. use a more restrictive definition of a model[5]:

*‘A model is a description of a (part of) systems written in a well-defined language. A well-defined language is a language with well-defined form (syntax), and meaning (semantics), which is suitable for automated interpretation by a computer.’*

In MDE, a software system is specified as a set of models that are repeatedly refined until a model is obtained that captures enough details such that it can be implemented. MDE combines process and analysis with architecture[6].

An important aspect of MDE is the emphasis it puts on bridges between technological spaces, and on the integration of bodies of knowledge developed by different research communities[7]. Different models can be used to express

different concerns, such as functionality, maintainability, etc. MDE emphasizes the need to have productive models that can be automatically manipulated by programs. To make the models productive, it is necessary to completely and formally define them[8].

In MDE a metamodel is constructing a formal definition of a modeling language, and thus describing the rules to which the structure of a model must adhere. Metamodels are expressed in metamodeling languages such as MOF[9] or Ecore[10]. When a model respects the metamodel structure, the model is said to be conform to the metamodel or an instance of the metamodel.

In this thesis, the used requirements models are realized according to the principles of MDE.

## 2.3 Requirements Engineering

Zave[11] defines the area of Requirements Engineering (RE) as:

*‘Requirements engineering is the branch of software engineering concerned with the real-world goals for, functions of, and constraints on software systems. It is also concerned with the relationship of these factors to precise specifications of software behavior and to their evolution over time and across software families.’*

Van Lamsweerde[12] summarizes the different activities performed in the area of RE as:

**Domain analysis:** Study of the environment in which the software is to be used. Identification of stakeholders and analysis of the current problems and opportunities and the resulting general objectives of the software.

**Elicitation:** Exploration of alternative models for the target system to meet the general objectives resulting from domain analysis. Components of the alternative models are identified, possibly with the help of hypothetical interaction scenarios.

**Negotiation and agreement:** Alternative requirements/assumptions are evaluated; risks are analyzed. Agreement between stakeholders is reached.

**Specification:** Requirements and assumptions are formulated in a precise way.

**Specification analysis:** Specifications are checked for deficiencies (e.g. inadequacy, incompleteness or inconsistency) and feasibility (in terms of resources required, development costs etc.)

**Documentation:** Various decisions made during the process are captured together with the underlying rationale and assumptions.

**Evolution:** The requirements are modified to accommodate corrections.

In this thesis the focus lies on the ‘Evolution’ activity in the RE in general and analysis of evolving requirements and determining the ripple effect caused by changes in particular.

### 2.3.1 Software Requirements

The oldest definition of a requirement is attributed to Ross and Schoman[13] who describe it as:

*‘requirements definition is a careful assessment of the needs that a system is to fulfill. It must say why a system is needed, based on current or foreseen conditions, which may be internal operations or an external market. It must say what system features will serve and satisfy this context. And it must say how the system is to be constructed.’*

In this thesis the working definition of a requirement is a general definition as described in SWEBOK[14]:

*‘A requirement is a property which must be exhibited by a system.’*

Requirements are categorized as functional and non-functional requirements by Sommerville[15] as follows:

**Functional system requirements:** These are system services which are expected by the user of the system. In general, the user is uninterested in how these services are implemented so the software engineer should avoid introducing implementation concepts in describing these requirements.

**Non-functional requirements:** These set out the constraints under which the system must operate and the standards which must be met by the delivered system.

In this thesis the use of ‘requirement’ without further classification indicates a functional requirement.

### 2.3.2 Requirements Traceability

General traceability is defined by the IEEE Standard Glossary of Software Engineering Terminology[16] 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’*

Traceability can be established between any artifacts produced during the software engineering process. Requirements may capture traceability information to refer to other artifacts produced during the software engineering process, or to other related requirements.

Götel and Finkelstein[17] define traceability in the context of RE as:

*‘Requirements traceability refers to the ability to follow the life of a requirement in a forward 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.)’*

The concepts of *forward* and *backward* direction of traceability have standard definitions in literature[18]:

**Forward traceability** is the ability to trace a requirement to components of a design or implementation.

**Backward traceability** is the ability to trace a requirement to its source, i.e. to a person, institution, law, argument, etc.

Traceability to and from requirements to other products of the development process are traditionally used to indicate that the resulting implemented systems meet the necessary capabilities described by contractual agreements. Traceability is also used to indicate that rationale behind design decisions are sound and thus used as a measure of system quality and software process maturity[19].

A distinction is made between how the requirement specification came to be, and how the requirement specification is used, so-called pre- and post-requirements specification traceability[17]:

**Pre-requirements specification traceability** refers to those aspects of a requirement's life prior to its inclusion in the requirements specification.

**Post-requirements specification traceability** refers to those aspects of a requirement's life that result from inclusion in the requirements specification.

The above mentioned definitions and categorizations of traceability concern the traceability between requirements and other (non-requirements) artifacts. To determine how separate requirements captured by a requirements specification relate to each other, traceability can be added between individual requirements. This led to the definitions of inter- and extra-requirements traceability[20]:

**Inter-requirements traceability** refers to the relationships between requirements.

**Extra-requirements traceability** refers to the relationships between requirements and other artifacts.

In this thesis the focus lies on post inter-requirements traceability.

### 2.3.3 Requirements traceability metamodel

By using inter-requirements traceability, traceability information can be captured in a requirements model which indicates relatedness between requirements. The requirements traceability can be used in development activities and decision making during software development, for example release planning, requirements validation, change impact analysis, testing and requirements reuse[21]. Each entity and link in the meta-model can be specialized and instantiated to create organization or project specific traceability models.

Ramesh and Jarke[22] provide a (simple) requirements traceability metamodel, which is depicted in Figure 2.1.

Within the presented research the focus lies on the trace relations between the objects that represent the physical requirements documentation.

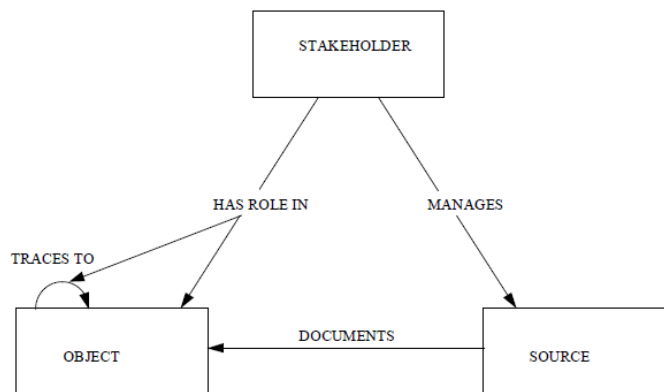


Figure 2.1: Traceability meta model[22]

## 2.4 Change Impact Analysis

A change of requirements affects the existing artifacts. Existing artifacts should be updated to reflect the changes. Software systems grow in size and complexity beyond the point where software engineers are able to comprehend them. As a result change impact analysis is performed to determine the requirements that require change to ‘counter’ the deterioration that occurs when changing requirements. Change Impact Analysis (CIA) is defined by Heindl[23] as:

*‘the activity where the impacts of a requirement’s change on other artifacts are identified’*

Bohner uses a similar working definition for CIA in [1]:

*‘The determination of potential effects to a subject system resulting from a proposed software change.’*

The set of artifacts affected by the initial or proposed change is categorized as the *starting impact set* (SIS). By performing trace-based change impact analysis the traceability links from the SIS are followed to other artifacts. The artifacts reached through the traceability links, which make up the *candidate impact set* (CIS), are considered candidates for change, and need to be checked to determine if they are affected. In the case that such artifact is affected, it is considered to be in the *actual impact set* (AIS). From an artifact just added to the AIS traceability links from that artifact need to be followed to determine if there are artifacts that should be added to the CIS. In this regard, the CIA process is iterative and explorative in nature.

Bohner categorizes these impacts as direct and indirect[1]:

**Direct impacts** occur when the artifact affected is reached by a direct trace relation from an artifact that is in the ‘starting impacted set’.

**Indirect impacts** occur when the artifact affected is reached by an acyclic path of trace relations from an (affected) artifact that is in the ‘candidate impacted set’.

Indirect impacts are also referred to by Bohner as an N-level impact where N is the number of intermediate traces between the artifact belonging to the SIS to the found impact.

The result of CIA is a change impact prediction (CIP), which is defined by Arnold and Bohner[24] as:

*‘Change impact prediction enumerates the set of artifacts to be affected by the change impact analysis.’*

The CIA is performed using the traceability information available. Performing CIA on a requirements model which uses traceability information that lacks additional semantics, the traditional predecessor-successor semantics are used. This will result in a reachability analysis. CIA indicates the set of requirements that is potentially affected by a requirement change, based on transitivity. There is no indication if a propagation occurs or not.

Consider the traceability matrices in Figure 2.2 and Figure 2.3. Figure 2.2 indicates the traceability relation between the different software requirements. An  $\times$  indicates a traceability relation from the requirement indicated in the row to the requirement indicated in the column. Figure 2.3 is the result after applying a reachability analysis. This matrix indicates from each requirement every other requirement can be reached by using the traceability information. This means that when performing a change impact prediction of a change on any given requirement in this model, the prediction would yield all requirements captured in this model as possibly impacted by the change.

|     | R1       | R2       | R3       | R4       | R5       | R6       | R7       | R8       | R9       | R10      |
|-----|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|
| R1  |          | $\times$ |          |          |          | $\times$ |          |          |          | $\times$ |
| R2  |          |          | $\times$ |          | $\times$ |          | $\times$ |          |          |          |
| R3  |          |          |          | $\times$ | $\times$ |          |          | $\times$ |          |          |
| R4  |          |          |          |          |          |          | $\times$ |          | $\times$ | $\times$ |
| R5  | $\times$ |          |          | $\times$ |          |          |          | $\times$ |          |          |
| R6  |          | $\times$ |          |          | $\times$ |          |          |          |          | $\times$ |
| R7  |          |          | $\times$ |          |          | $\times$ |          |          |          | $\times$ |
| R8  |          | $\times$ |          | $\times$ |          |          |          |          | $\times$ |          |
| R9  |          |          | $\times$ |          | $\times$ |          |          |          |          | $\times$ |
| R10 |          | $\times$ |          |          | $\times$ |          |          | $\times$ |          |          |

Figure 2.2: Traceability matrix indicating trace relations

The assumption that the likelihood of a propagated change decreases when the level of indirect impact becomes higher are not guided by semantics[1]. To counter the problem of the explosion of impacts, Bohner states that additional semantics must be employed to perform more precise CIA.

|            | <b>R1</b> | <b>R2</b> | <b>R3</b> | <b>R4</b> | <b>R5</b> | <b>R6</b> | <b>R7</b> | <b>R8</b> | <b>R9</b> | <b>R10</b> |
|------------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|------------|
| <b>R1</b>  |           | 1         | 2         | 3         | 2         | 1         | 2         | 2         | 3         | 1          |
| <b>R2</b>  | 2         |           | 1         | 2         | 1         | 2         | 1         | 2         | 3         | 2          |
| <b>R3</b>  | 2         | 2         |           | 1         | 1         | 3         | 2         | 1         | 2         | 2          |
| <b>R4</b>  | 3         | 2         | 2         |           | 2         | 2         | 1         | 2         | 1         | 1          |
| <b>R5</b>  | 1         | 2         | 3         | 1         |           | 2         | 2         | 1         | 2         | 2          |
| <b>R6</b>  | 2         | 1         | 2         | 2         | 1         |           | 2         | 2         | 3         | 1          |
| <b>R7</b>  | 3         | 2         | 1         | 2         | 2         | 1         |           | 2         | 3         | 1          |
| <b>R8</b>  | 3         | 1         | 2         | 1         | 2         | 3         | 2         |           | 1         | 2          |
| <b>R9</b>  | 3         | 2         | 1         | 2         | 1         | 3         | 3         | 2         |           | 1          |
| <b>R10</b> | 2         | 1         | 2         | 2         | 1         | 3         | 2         | 1         | 2         |            |

Figure 2.3: Trace reachability matrix with distance indicator

## 2.5 Conclusion

This chapter described the basic concepts and background of the research presented in this thesis. For the different fields of research, the work performed in this thesis is positioned. The research in this thesis focuses on providing tool support for performing change impact analysis of evolving requirements models, using inter-requirements traceability.

For this research to be placed in the broader context of MDE, well defined syntax and semantics are required for the CIA on requirements models.





## Chapter 3

# Formalized requirements & relations

### 3.1 Introduction

The main goal of this work is to provide tool support for performing CIA on inter-requirements models, using the semantics of trace relations. Research performed within the QuadREAD project, which is a joint research project by the University of Twente and various business partners, yielded a requirements metamodeling approach proposed by Göknil et al.[2]. This metamodeling approach contains well-defined semantics for requirements and requirement relations. When *semantics of requirements*, *formalized requirements* or *semantics of requirements relations* is used, they refer to the proposed formalization by Göknil et al. The relations captured by the metamodel are requirement relations found in literature. This semantics of requirement relations will be used as traceability information to perform CIA.

In section 3.2 the requirements metamodel is described. Section 3.3 describes the formalization of requirements and section 3.4 describes the formalization of the requirement relations. This chapter is concluded in section 3.5.

### 3.2 Requirements metamodel

The requirements metamodel captures formal requirement relationship types. These formal requirements relations allow for reasoning about requirements, such as inferencing and consistency checking. The requirements metamodel is based on a review of literature. The proposed requirements metamodel is depicted in Figure 3.1. Additional entities that are not related to the requirement relations formalization, such as stakeholder and testcases have been left out for overview purposes.

The metamodel has a ‘RequirementsModel’ as main entity, which captures zero or more requirements. Each ‘Requirement’ has a number of attributes and is identified by an ID. Requirements can be related to each other through ‘Relationship’. The metamodel describes five relationship types. These relations have the following informal definitions in literature[2]:

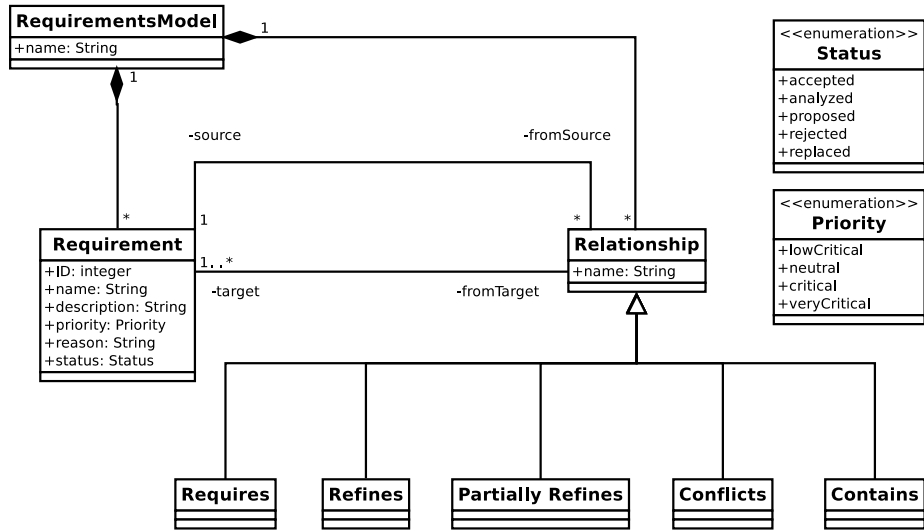


Figure 3.1: Requirements metamodel with formal definitions[2]

**Contains:** This relationship enables a complex requirement to be decomposed into parts. A composite requirement may state that the system shall do A and B and C, which can be decomposed into the requirements that the system shall do A, the system shall do B, and the system shall do C.

**Refines:** A requirement  $R_1$  *refines* a requirement  $R_2$  if  $R_1$  is derived from  $R_2$  by adding more details to it. The refined requirement can be seen as an abstraction of the detailed requirements.

**Partially refines:** A requirement  $R_1$  *partially refines* a requirement  $R_2$  if  $R_1$  is derived from  $R_2$  by adding more details to parts of  $R_2$  and excluding the unrefined parts of  $R_2$ . This relation can be described as a special combination of decomposition and refinement.

**Requires:** A requirement  $R_1$  *requires* requirement  $R_2$  if  $R_1$  is fulfilled only when  $R_2$  is fulfilled. The required requirement can be seen as a precondition for the requiring requirement.

**Conflicts:** A requirement  $R_1$  *conflicts* with a requirement  $R_2$  if the fulfillment of  $R_1$  excludes the fulfillment of  $R_2$  and vice versa.

### 3.3 Formalization of requirements

In the metamodel depicted in Figure 3.1, the requirement is defined by Göknil et al. in first order logic (FOL). This formalization follows from the general notion of a requirement as described by the Software Engineering Body Of Knowledge (SWEBOK)[14] and is defined as follows[2]:

Requirement  $R$  is a tuple  $\langle P, S \rangle$  where  $P$  is a predicate (the property or properties) and  $S$  is the set of systems that satisfy  $P$ , i.e.  $\forall s \in S : P(s)$

Predicate  $P$  can be represented in a conjunctive normal form (CNF) as follows:

$P = p_1 \wedge p_2 \wedge \dots \wedge p_{n-1} \wedge p_n$ ; where  $n \geq 1$  and  $p_n$  is the disjunction of literals. From here on the CNF will be written as  $(p_1 \dots p_n)$ .

In this formalization  $P$  is the intentional definition of the requirement and  $S$  is the extensional definition of the requirement.

### 3.4 Formalization of requirement relations

Here the formalization of each of the requirement relationships are presented. For example use of these requirement relations, see Göknil et al.[2] or the provided tutorial for requirements relations[25].

#### 3.4.1 Contains

Let  $R_1 = \langle P_1, S_1 \rangle, R_2 = \langle P_2, S_2 \rangle, \dots, R_k = \langle P_k, S_k \rangle$  be requirements where  $k \geq 2$ .  $P_2, P_3, \dots, P_k$  are formulas in CNF as follows:

$$P_i = (p_1^i \dots p_{m_i}^i); m_i \geq 1, i \in \{2, \dots, k\}$$

$R_1$  contains  $R_2, \dots, R_k$  iff  $P_1$  is derived from  $P_2, P_3, \dots, P_k$  as follows:

$$P_1 = P_2 \wedge P_3 \wedge \dots \wedge P_k \wedge P'$$

where  $P'$  denotes properties that are not captured in  $P_2, P_3, \dots, P_k$ . Thus Therefore,  $S_1 \subset S_2, S_1 \subset S_3, \dots, S_1 \subset S_k$ .

Note: Complete decomposition is not assumed. It also does not mean that  $P_2$  and  $P_3$  are disjunct.

#### 3.4.2 Refines

Let  $R_1 = \langle P_1, S_1 \rangle$  and  $R_2 = \langle P_2, S_2 \rangle$  be requirements.  $P_1$  and  $P_2$  are formulas and the CNF of  $P_2$  is:

$$P_2 = (p_1 \dots p_n) \wedge (q_1 \dots q_m); n \geq 1, m \geq 0$$

Let  $p'_1, p'_2, \dots, p'_{n-1}, p'_n$  be the disjunction of literals such that  $p'_i \rightarrow p_i$ , for  $i \in \{1, \dots, n\}$ , and the following statements hold:

1.  $P_1 = (p'_1 \dots p'_n) \wedge (q_1 \dots q_m); n \geq 1, m \geq 0$
2.  $\exists s \in S_2 : s \notin S_1$

Then  $R_1$  refines  $R_2$ .

#### 3.4.3 Partially Refines

Let  $R_1 = \langle P_1, S_1 \rangle$  and  $R_2 = \langle P_2, S_2 \rangle$  be requirements.  $P_1$  and  $P_2$  are formulas and the CNF of  $P_2$  is:

$$P_2 = (p_1 \dots p_n) \wedge (q_1 \dots q_m); m, n \geq 1$$

Let  $q'_1, q'_2, \dots, q'_{m-1}, q'_m$  be the disjunction of literals such that  $q'_i \rightarrow q_i$ , for  $i \in \{1, \dots, m\}$  and the following statements hold:

1.  $P_1 = q'_1 \wedge q'_2 \wedge \dots \wedge q'_{m-1} \wedge q'_m$
2.  $\exists s \in S_2 : s \notin S_1, \exists s \in S_1 : s \notin S_2$  and  $\exists s \in S_1 \cap S_2$

Then  $R_1$  *partially refines*  $R_2$ .

### 3.4.4 Requires

Let  $R_1 = \langle P_1, S_1 \rangle$  and  $R_2 = \langle P_2, S_2 \rangle$  be requirements. Then  $R_1$  *requires*  $R_2$  when:

1.  $\forall s \in S_1 : s \in S_2$  and
2.  $\exists s \in S_2 : s \notin S_1$

### 3.4.5 Conflicts

Let  $R_1$  and  $R_2$  be requirements such that  $R_1 = \langle P_1, S_1 \rangle$  and  $R_2 = \langle P_2, S_2 \rangle$ .  $R_1$  *conflicts*  $R_2 \leftrightarrow \neg \exists s : s \in S_1 \wedge s \in S_2 : P_1(s) \wedge P_2(s)$ .

This means that  $S_1 \cap S_2 = \emptyset$ .

## 3.5 Conclusion

As mentioned in section 3.3, the formalization of requirements is defined in both intentional terms (over the properties captured by the requirement) and extensional terms (over the set of systems that satisfy the requirement).

All requirement relations are defined of the set of systems that satisfy the requirement and are expressed over the extensional definitions of the requirement.

Additionally, for some requirement relationships (contains, refines and partially refines) the formalization is expressed using relations over the intentional definitions of the requirement, e.g. the properties captured by the related requirements.

## Chapter 4

# Change classification with formal semantics

### 4.1 Introduction

Requirements are subject to change from the moment that they are elicited. To determine the granularity of change, the structure of requirements needs to be known. Granularity of change alone is not sufficient to determine the impact on related requirements. Additional information concerning the rationale of change is needed. By identifying the change source, the rationale of why a requirement is being changed, is determined.

In section 4.2 literature on structure of textual requirements is presented. By identifying parts that can be changed in the textual requirement, the granularity of change is determined. These parts of the textual requirement are mapped onto the formalized requirement of Chapter 3. From this mapping the classification of change for the formalized requirement is determined.

In section 4.3 the change classification for the formalized requirements are defined in FOL.

In section 4.4 literature on classification of requirement evolution is presented. Following the classification of requirement evolution, two types of change rationales are identified and formalized.

In section 4.5 this chapter is summarized and concluded.

### 4.2 Structure of requirements

The formalized requirement in the requirement model represents the textual requirement from the requirements specification document. To determine in which way the requirement in the model can be changed, first the structure of the textual requirement is investigated. By determining the structure of a textual requirement, the various elements that compose a requirement are identified. In literature, textual requirements are composed of elements of increasingly finer granularity. The identified elements in textual requirements are then mapped onto the formalized requirements.

### 4.2.1 Textual requirements

Heninger explicitly mentions six requirements which a software requirements document should satisfy[26]:

1. It should only specify external system behavior
2. It should specify constraints on the implementation
3. It should be easy to change
4. It should serve as a reference tool for system maintainers
5. It should record forethought about the life-cycle of the system
6. It should characterize acceptable responses to undesired events

Although requirements 4-6 can be regarded as non-functional (quality) requirements for the requirements document, requirements 1 and 2 explicitly mention the *external behavior* and *constraints on this behavior* respectively. These are both reflected in the actual requirements in the document. This indicates that elements of different granularities are present in requirements as a whole.

Wasson further refines this as to how requirements are structured. He states that a textual requirement should be interpreted by identifying key elements of the requirement, the so-called *requirement primitives*. These *requirement primitives* compose the textual requirements. The different types of requirement primitives described by Wasson are the following[3]:

- Capability to be provided
- Relational operators
- Boundary constraints, thresholds, tolerances or conditions (limitations)

Each requirement describes a *capability* that the system should provide. This is the main functionality that should be provided. The functionality can be further refined by adding additional information which makes the *capability* more specific. This is done by setting *limitations* on the capability, certain *thresholds* or other limitations such as *tolerances*, *conditions* or *boundary constraints*. Compared to Heninger Wasson explains in further detail as to how the *limitations* are related to the capability. This is through the *relational operator*, which describes *how* the added additional information is related to the main capability.

Using Wasson's primitives, the part-whole composition of the requirement in the textual domain in an UML diagram is presented in Figure 4.1.

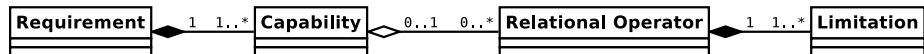


Figure 4.1: Textual requirement structure using Wasson's primitives

From this composition it is interpreted that requirements describe capabilities that the implemented system should provide. These capabilities can be

described more precisely by adding limitations. Limitations are bound to the capability by a relational operator.

To illustrate the structure of a textual requirement using Wasson's primitives, consider the following example:

**Example: decomposition of textual requirement**

*R<sub>97</sub>: 'The system shall allow only the administration to manage courses'*

Using Wasson's requirement primitives and terms of *capabilities*, *limitations* and *relational operators*, requirements structure can be determined. The capability that this requirement describes is that the system should provide the functionality of managing courses. This functionality is limited in such a way that this capability is only provided to the administration. The requirement is also clear as to how this limitation is imposed on the capability, by means of permission based on usertype, which is thus the relational operator.

Decomposition of requirement using Wasson's primitives:

**Capability:** The system shall [provide functionality of] managing courses

**Relational operator:** Limited by user type

**Limitations:** Only by the administration

The glossary of the requirements specification document states that 'managing' is defined as the operations 'creating', 'reading', 'updating', and 'deleting', and thus can be regarded as four separate capabilities.

### 4.2.2 Formalized requirements

The definition used by Göknil et al. is that a textual requirement is a description of a system *property* or system *properties* which need to be fulfilled[2].

The notion of Göknil's *property* corresponds to Wasson's *capability*. The *property* can be changed in such a way that the set of systems satisfying the changed property is changed, while the main capability remains the same. Following Wasson's primitives it is interpreted that the *property* of the formalized requirement captures may capture elements of finer granularity. These elements are classified as *constraints*.

The composition for the formalized requirement including the 'constraint' elements is presented in an UML diagram, depicted in Figure 4.2.



Figure 4.2: Requirement structure for the formalized requirement

Thus, a requirement is expressed by the set of *properties* which the system should provide, which can be expressed in more detail by contained *constraints*.

Using this composition of the formalized requirement, consider the example requirement:

**Example: decomposition of formalized requirement**

$R_{97}$ : ‘The system shall allow only the administration to manage courses’

Using the formalization of requirements, the requirement is defined as follows:

$$R_{97} = \langle P_{97}, S_{97} \rangle$$

$$P_{97} = P_1 \wedge P_2 \wedge P_3 \wedge P_4$$

Such that property  $P_{97}$  is decomposed into four properties  $P_1$ ,  $P_2$ ,  $P_3$  and  $P_4$  with:

Using formalized requirements:

**Property  $P_1$ :**  $enable(x_1, y) \wedge create(x_1) \wedge course(y) \wedge allow(x_1, z) \wedge administrator(z) \wedge \neg lecturer(z) \wedge \neg student(z)$

**Constraint in  $P_1$ :**  $allow(x_1, z) \wedge administrator(z) \wedge \neg lecturer(z) \wedge \neg student(z)$

**Property  $P_2$ :**  $enable(x_2, y) \wedge read(x_2) \wedge course(y) \wedge allow(x_2, z) \wedge administrator(z) \wedge \neg lecturer(z) \wedge \neg student(z)$

**Constraint in  $P_2$ :**  $allow(x_2, z) \wedge administrator(z) \wedge \neg lecturer(z) \wedge \neg student(z)$

**Property  $P_3$ :**  $enable(x_3, y) \wedge update(x_3) \wedge course(y) \wedge allow(x_3, z) \wedge administrator(z) \wedge \neg lecturer(z) \wedge \neg student(z)$

**Constraint in  $P_3$ :**  $allow(x_3, z) \wedge administrator(z) \wedge \neg lecturer(z) \wedge \neg student(z)$

**Property  $P_4$ :**  $enable(x_4, y) \wedge delete(x_4) \wedge course(y) \wedge allow(x_4, z) \wedge administrator(z) \wedge \neg lecturer(z) \wedge \neg student(z)$

**Constraint in  $P_4$ :**  $allow(x_4, z) \wedge administrator(z) \wedge \neg lecturer(z) \wedge \neg student(z)$

With the following predicate symbols:

$enable(x, y)$ : stating that object  $x$  is provided over  $y$  by the system.

$create(x)$ : stating that  $x$  is the ‘creating’ operation.

$read(x)$ : stating that  $x$  is the ‘reading’ operation.

$update(x)$ : stating that  $x$  is the ‘updating’ operation.

$delete(x)$ : stating that  $x$  is the ‘deleting’ operation.

$course(x)$ : stating that  $x$  is a course.

$allow(u, x)$ : stating that object  $x$  is allowed to  $u$ .

$administrator(x)$ : stating such that  $x$  is an administrator user.

$lecturer(x)$ : stating that  $x$  is a lecturer user.

$student(x)$ : stating that  $x$  is a student user.



### 4.3 Semantics of requirement changes

Following such composition of the formalized requirement, in *properties* which can capture *constraints*, a classification of changes for the requirement model can be determined. This is done by applying the basic operations of ‘create’, ‘read’, ‘update’, ‘delete’ (CRUD)[27] on the different elements of the requirement model. The requirement model consists of *requirements* and *relations*. The changes are listed below.

#### Requirement changes

- Add requirement
- Delete requirement
- Update requirement
  - Add property to requirement
  - Delete property from requirement
  - Change property in requirement
  - Add constraint to requirement
  - Delete constraint from requirement
  - Change constraint in requirement

#### Requirement relation changes

- Add requirement relation
- Delete requirement relation
- Update requirement relation
  - Change directionality of requirement relation
  - Change requirement relation type
  - Change directionality and relationship type

Elementary requirement changes to the requirements model are formalized below. To denote change, the  $\mapsto$  symbol is used, in the following way:  $R \mapsto R^l$  to denote a change in general, where  $R$  is the requirement before the change and  $R^l$  is the requirement after the change. In the same fashion, change is indicated for properties and sets of systems. Specific kind of changes are denoted by denoting the type of change over the  $\mapsto$  symbol.

#### 4.3.1 Update requirement

Update a requirement  $R = \langle P, S \rangle$  in one of the following ways:

- By adding a property  $pt$  to the requirement  $R$ , denoted as  $R \xrightarrow{+pt} R^l$ .
- By deleting a property  $pt$  from the requirement  $R$ , denoted as  $R \xrightarrow{-pt} R^l$ .

- By changing a property  $pt$  captured by the requirement  $R$ , denoted as  $R \xrightarrow{pt \rightarrow pt^l} R^l$ .
- By adding a constraint  $ct$  to an existing property  $pt$  of the requirement  $R$ , denoted as  $R \xrightarrow{+ct} R^l$ .
- By deleting a constraint  $ct$  from an existing property  $pt$  of the requirement  $R$ , denoted as  $R \xrightarrow{-ct} R^l$ .
- By changing a constraint  $ct$  captured by requirement  $R$ , denoted as  $R \xrightarrow{ct \rightarrow ct^l} R^l$ .

### Add property $pt$ to requirement $R$

Adding a property  $pt$  to a requirement such that the requirement expresses the added functionality. The requirement is changed such that it also captures the newly added property  $pt$ . This is formalized as follows:

Let  $R = \langle P, S \rangle$  be the requirement before adding the property  $pt$ , and let  $R^l = \langle P^l, S^l \rangle$  be the requirement after adding the property  $pt$ , where  $P$  is a formula in CNF as follows:

- $P = (p_1 \cdots p_i); i \geq 1$

Then  $R \xrightarrow{+pt} R^l$  iff  $P^l$  is derived from  $P$  as follows:

- $P^l = P \wedge P_{pt}$ , where  $P_{pt}$  denotes the properties that are captured in  $pt$

The set of systems  $S^l$  not only satisfies the previously captured property  $P$  but also the newly added property  $P^l$ , thus the set of systems satisfying  $P^l$  is such that  $S^l \subset S$ .

### Delete property $pt$ from requirement $R$

Deleting a property  $pt$  from a requirement is such that some functionality is removed from the requirement and thus no longer expressed by the requirement. The requirement is changed such that the removed property  $pt$  is no longer captured by the requirement. This is formalized as:

Let  $R = \langle P, S \rangle$  be the requirement before deleting the property  $pt$ , and let  $R^l = \langle P^l, S^l \rangle$  be the requirement after deleting the property  $pt$ , where  $P$  is a formula in CNF as follows:

- $P = (p_1 \cdots p_n) \wedge (q_1 \cdots q_m); n, m \geq 1$

Then  $R \xrightarrow{-pt} R^l$  iff  $P^l$  is derived from  $P$  as follows:

- $P^l = (p_1 \cdots p_n); n \geq 1$ , where  $(q_1 \cdots q_m); m \geq 1$  denotes properties that are captured in  $pt$  and removed.

The set of systems  $S^l$  is such that  $S^l \supset S$ .

**Change property  $pt$  in requirement  $R$  with property  $pt^l$** 

Changing a property  $pt$  to  $pt^l$  in a requirement is a composite change. The change is composed of ‘delete property  $pt$ ’ from the requirement and ‘add property  $pt^l$ ’ to the requirement. Effectively the deleted property is replaced by another property. Due to the explicit nature of relation between the deleted property and the added property by means of replacement, the composite operation of ‘change property’ is given. This is formalized as:

Let  $R = \langle P, S \rangle$  be the requirement before changing the property  $pt$  with the property  $pt^l$ , and let  $R^l = \langle P^l, S^l \rangle$  be the requirement after changing the property  $pt$  with property  $pt^l$ , where  $P$  is a formula in CNF as follows:

- $P = (p_1 \cdots p_v) \wedge (q_1 \cdots q_w); v \geq 1, w \geq 0$

Then  $R \xrightarrow{pt \rightarrow pt^l} R^l$  iff  $P^l$  is derived from  $P$  as follows:

- $P^l = (t_1 \cdots t_z) \wedge (q_1 \cdots q_w); z \geq 1, w \geq 0$ , where  $(p_1 \cdots p_v); n \geq 1$  denotes properties that are captured in  $pt$  and  $(t_1 \cdots t_z); z \geq 1$  denotes properties that are captured in  $pt^l$ .

For this change, a subset of superset relation from the sets of systems before and change can not be guaranteed.

**Add constraint  $ct$  to property  $pt$  of requirement  $R$** 

Adding a constraint  $ct$  to a property  $pt$  indicates that by adding more detail to the property, thus it becomes more specific. This is formalized as follows:

Let  $R = \langle P, S \rangle$  be the requirement before adding constraint  $ct$  to the property  $pt$ , and let  $R^l = \langle P^l, S^l \rangle$  be the requirement after adding the constraint  $ct$  to the property  $pt$  with property  $pt^l$ , where  $P$  is a formula in CNF as follows:

- $P = (p_1 \cdots p_n) \wedge (q_1 \cdots q_m); n \geq 1, m \geq 0$

Let  $p_1^l, p_2^l, \dots, p_{n-1}^l, p_n^l$  be the disjunction of literals such that  $p_j^l \rightarrow p_j$  for  $j \in \{1, \dots, n\}$ .

Then  $R \xrightarrow{+ct} R^l$  iff  $P^l$  is derived from  $P$  by replacing every  $p_j$  in  $P$  with  $p_j^l$  for  $j \in \{1, \dots, n\}$  such that the following two statements hold:

1.  $P^l = (p_1^l \cdots p_n^l) \wedge (q_1 \cdots q_m); n \geq 1, m \geq 0$
2.  $\exists s \in S : s \notin S^l$

**Delete constraint  $ct$  from property  $pt$  of requirement  $R$** 

By deleting a constraint  $ct$  from a property  $pt$ , detail is removed from the requirement. This means that the requirement becomes less stringent, and the property more abstract. This is formalized as follows:

Let  $R = \langle P, S \rangle$  be the requirement before deleting the constraint  $ct$  from the property  $pt$ , and  $R^l = \langle P^l, S^l \rangle$  be the requirement after deleting the constraint  $ct$  from the property  $pt$ , where  $P$  is a formula in CNF as follows:

- $P = (p_1^l \cdots p_n^l) \wedge (q_1 \cdots q_m); n \geq 1, m \geq 0$

Let  $p_1, p_2, \dots, p_{n-1}, p_n$  be the disjunction of literals such that  $p_j^l \rightarrow p_j$  for  $j \in \{1, \dots, n\}$ .

Then  $R \xrightarrow{ct} R^l$  iff  $P^l$  is derived from  $P$  by replacing every  $p_j^l$  in  $P$  with  $p_j$  for  $j \in \{1, \dots, n\}$  such that the following two statements hold:

1.  $P^l = (p_1 \cdots p_n) \wedge (q_1 \cdots q_m); n \geq 1, m \geq 0$
2.  $\exists s \in S^l : s \notin S$

#### Change constraint $ct$ of the requirement $R$ with constraint $ct^l$

Similar to changing a property, changing a constraint is a composite change of replacing one constraint by another. It is formalized as follows:

Let  $R = \langle P, S \rangle$  be the requirement before changing the constraint  $ct$  with the constraint  $ct^l$ , and  $R^l = \langle P^l, S^l \rangle$  be the requirement after changing the constraint  $ct$  with the constraint  $ct^l$ , where  $P$  is a formula in CNF as follows:

- $P = (p_1 \cdots p_n) \wedge (q_1 \cdots q_m); n, m \geq 1$

Then  $R \xrightarrow{ct \rightarrow ct^l} R^l$  iff  $P^l$  is derived from  $P$  as follows:

- $P^l = (t_1 \cdots t_z) \wedge (q_1 \cdots q_m); m, z \geq 1$ , where  $(p_1 \cdots p_n); n \geq 1$  denotes constraints that are captured in  $ct$  and  $(t_1 \cdots t_z); z \geq 1$  denotes constraints that are captured in  $ct^l$

## 4.4 Semantics of change rationale

Semantics of requirement changes as described in section 4.3 do not elaborate on why a change needs to be performed on the requirements model. To identify what the cause of requirements models change is, literature is consulted to determine reasons of software evolution.

The following sources of software evolution are identified[28][29]:

**Domain** covers the model of the real world considered by the system, i.e., the environment. Any change in that model may force the system change. Changes originating from this source are considered ‘adaptive maintenance’.

**Experience** The users of the system gain experience over time and they may require some improvements for the system, which in turn may cause the system to evolve. These changes are considered ‘perfective maintenance’.

**Process** includes the organizations and methods that may also impact the system and cause it to change. These changes are considered ‘corrective maintenance’.

Within the scope of requirements engineering, Van Lamsweerde introduces requirement description qualities such as good structuring and modifiability [30]. The requirements engineer may change the requirements model to improve the quality of the requirements descriptions. This is analog to ‘perfective maintenance’ found in software evolution. Changes to requirement models caused by perfective maintenance are considered ‘model refactoring’ in this thesis.

Evolution of requirements also leads to changes to the requirements model. This Van Lamsweerde names these changes as domain changes [30]. Domain changes can be regarded as ‘adaptive maintenance’ of the requirements model.

In this thesis, changes to the requirements model as a result from changes in the implementation are not considered.

#### 4.4.1 Formalization of requirements model

Individual requirements and relations are formalized as described in Chapter 3, but does not describe the formalization of the requirements model. The requirement model consists of the set of requirements captured by the model. Requirements are expressed using properties and the systems satisfying these properties. The model can therefore be represented as the set of all captured properties and the resulting satisfying system. The requirements model RM is formalized as follows:

- $RM$  is a collection of requirements  $R_1 = \langle P_1, S_1 \rangle, R_2 = \langle P_2, S_2 \rangle, \dots, R_k = \langle P_k, S_k \rangle$ , with  $k \geq 1$ .
- $RM$  is a tuple  $\langle P_{RM}, S_{RM} \rangle$ , with:
  - $P_{RM} = P_1 \wedge P_2 \wedge \dots \wedge P_k$ , the set of properties captured by the requirements in the requirements model
  - $S_{RM} = S_1 \cap S_2 \cap \dots \cap S_k$ , the set of systems that satisfy  $P_{RM}$  such that  $\forall s \in S_{RM} : P_{RM}(s)$
- Thus  $P_{RM}$  can be represented in a conjunctive normal form (CNF) in the following way:  $P_{RM} = (p_1 \dots p_n)$

Using this formalization for requirement model, both types of considered change rationales for requirement model changes, refactoring and domain change are formalized.

#### 4.4.2 Formalization of refactoring

Refactoring is changing the model without modifying overall system properties described in the model, in order to improve structuring of the model. The requirements model after change still reflects the domain. This is illustrated in Figure 4.3.

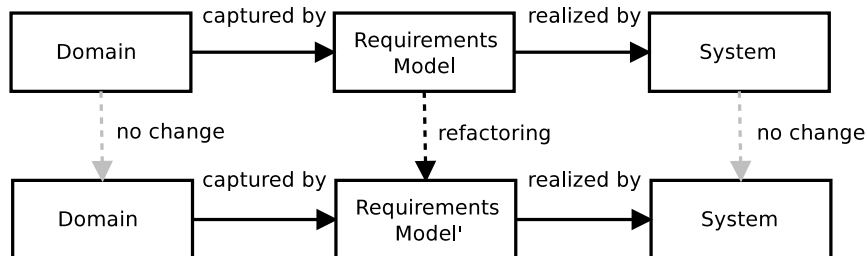


Figure 4.3: Overview of refactoring the requirements model

Refactoring is formalized as follows:  $RM \xrightarrow{\text{refactoring}} RM^l$  denotes a series of changes for model refactoring where  $RM = \langle P_{RM}, S_{RM} \rangle$  is the requirements model RM before the refactoring, and  $RM^l = \langle P_{RM}^l, S_{RM}^l \rangle$  is the requirements model RM after the refactoring.  $P_{RM}$  and  $P_{RM}^l$ , where are described in the following way:

1.  $P_{RM} = P_{RM}^l = (p_1 \cdots p_n)$ , with  $n \geq 1$  and  $p_n$  the disjunction of literals
2.  $S_{RM} = S_{RM}^l$

### 4.4.3 Formalization of domain change

Domain changes are changes to the requirements model in order to modify the overall system properties described by the model. Changes to the model caused by domain changes affect the properties described in the whole requirements model. When a property is removed as a result of a domain change, all occurrences of this property in the model should be deleted from the model. This is illustrated in Figure 4.4.

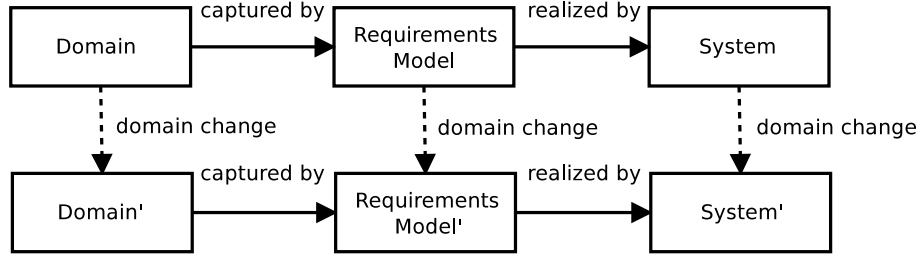


Figure 4.4: Overview of domain change

Domain changes are formalized as follows:

- $RM \xrightarrow{\text{domain change}} RM^l$  denotes a series of changes caused by domain change where
- $RM = \langle P_{RM}, S_{RM} \rangle$  is the requirements model RM before the domain change
- $RM^l = \langle P_{RM}^l, S_{RM}^l \rangle$  is the requirements model RM after the domain change

Then:

1.  $\neg \text{equals}(P_{RM}, P_{RM}^l)$ , where two formulas are equal when they have the same predicate symbols and arguments
2.  $S_{RM} \neq S_{RM}^l$ , where  $S_{RM} \neq \emptyset$

## 4.5 Conclusions

From the structure of textual requirements as described in section 4.2 using Wasson's primitives, the granularity of change for the formalized requirement has been determined in section 4.2.2.

The decomposition of formal requirements allows for the classification of requirement model changes as listed in section 4.3. Two types of requirement model changes have been identified; changes to requirements and changes to requirements relations. The requirements change types are then formalized in FOL.

In section 4.4 the different rationales that can be attributed to as why a requirement is changed are described as found in literature. For the rationales within the scope of this research, namely refactoring and domain changes, the formalization of these change types are described in subsections 4.4.2 and 4.4.3.

Now that changetypes and rationales of change have been identified and formalized, changes can be mapped onto requirement model changes that take into account requirements related by requirement relationships. This way changes the rationale of domain changes and the requirements relation can be used to determine if/how related requirements are impacted.





## Chapter 5

# Change propagation & consistency checking

### 5.1 Introduction

Classification and formalization of change presented in the previous chapter allow identification of change impacts. By using the requirements relations of the requirements metamodel as traceability information, CIA can be performed. With well-defined semantics of change classification, requirements and relations, propagation of change can be derived.

In section 5.2 the working definition of impact is presented. From this definition of impact, domain changes are used to determine the propagation of change for formalized requirement changes. With this formalization an exhaustive case study for each change type together with each type of requirements relation is performed to derive impact alternatives for each case. The change impact alternatives can be used to provide change impact predictions.

Section 5.3 describes how consistency checking can be performed for multiple impacts on the same requirement.

In section 5.4 taken approach is discussed. This chapter is then concluded in section 5.5.

### 5.2 Change propagation

#### 5.2.1 Impact

From the definitions of Bohner and Heindl used in section 2.4, *impact* can be interpreted as the *effect* that a change has on other software artifacts. By changing software artifacts, without updating the traceability information, deterioration occurs. Therefore trace information should be updated in order to prevent this deterioration. Thus the *impact* is directly related to updating software artifacts in order to prevent deterioration. The working definition for *impact* is:

**Impact** is the *needed change* of software artifacts caused by a change made in software artifacts.

Consider the semantics for refactoring and domain changes as described in section 4.4. The set of changes considered as refactoring does not change the set of properties that is captured by the requirements model, nor does it change the set of systems satisfying these requirement changes. Derived from the working definition of *impact*, which considers the *needed change* to cause the change propagation, it is determined that *refactoring* does not result in *needed changes* to be propagated.

The set of changes captured by a domain change however, do change the set of properties captured by the requirements model and the systems satisfying them. Domain changes are considered changes that may cause impact on related requirements. As a result change propagation is determined for domain changes.

### 5.2.2 Rules for impact propagation

In the requirements model, both requirements and relations are regarded as software artifacts. Therefore impact over a relation that results from a requirement change is indicated for each requirement relation by:

1. Impact on requirement
2. Impact on requirements relation

To determine the rules for change propagation, the semantics for domain change is considered. The given definition of domain change is such that  $RM = \langle P_{RM}, S_{RM} \rangle$  before the domain change, and  $RM^l = \langle P_{RM}^l, S_{RM}^l \rangle$  such that  $RM \neq RM^l$ . Thus when the requirements model is affected by a domain change,  $RM \neq RM^l$  must be ensured.

Semantics of requirements relations as described in Chapter 3 provides additional information about what properties related requirements capture and which sets of systems are satisfied by them. By using the requirements relations, it can be determined if the formalization for domain change holds.

Moreover, relations can become invalid due to change of a requirement. Although the formalization of domain change holds, the relation becomes invalid if the change is not propagated to the related requirement. In this situation, the requirements engineer can decide to remove the relation or to update the related requirement. In the latter case the relation is maintained.

It is derived that a propagation of change occurs when:

1.  $RM \neq RM^l$  does not hold
2. Although (1) is satisfied the requirements relation becomes invalid, and propagation of the change can prevent this

Here (1) is a case that must be fulfilled, where case (2) poses the choice to either propagate the change over the relation, or delete the requirements relation.

### 5.2.3 Implications of domain change

Model changes that reflect domain changes imply additional information about the change. These are made explicit:

- a. **add property to requirement:** The added property is not already captured by the model.
- b. **add constraint to property in requirement:** The property to which the constraint is added does not capture that or a more stringent constraint anywhere else in the model.
- c. **remove property from requirement:** The removed property should be removed from all (related) requirements capturing this property.
- d. **remove constraint from property in requirement:** The removed constraint from a certain property should be removed from all (related) occurrences of that property in the model.
- e. **change property in requirement:** All (related) occurrences of the property that is changed should be changed.
- f. **change constraint in property in requirement:** The changed constraint of a certain property should be changed in all (related) occurrences of that property in the model.
- g. **delete relation:** The (current) requirement relation has become invalid, but gives no indication if other requirements relations are applicable.

#### 5.2.4 Change impact propagation

The needed and possible propagated change can be determined when considering the following:

- Requirement change reflects a domain change
- Classification of change
- Semantics of requirements relation to related requirement

An systematic case analysis is performed to determine the propagation alternatives for each combination of change type, requirements relation and direction of propagation. The change impact alternatives for each of the combinations of change types with requirements relations are denoted in Tables 5.1 and 5.2. Each impact alternative is denoted as a tuple  $(a,b)$ , with  $a$  denoting the impact on the related requirement and  $b$  denoting the impact on the requirements relation. Multiple alternatives are separated by the logical OR symbol |. For brevity, NI indicates ‘no impact’ and DR indicates ‘delete relationship’.

The tables are followed by two illustrations of a single case analysis.

| Change   | Case 1  | Case 2   | Case 3   |
|--|---|--|--|
|  | $R_1 \xrightarrow{\text{cont}} R_2$   | $R_1 \xrightarrow{\text{ref}} R_2$   | $R_1 \xrightarrow{\text{p.ref}} R_2$   |
| a. add $R_x$                                     | (NI, NI)  | (NI, NI)   | (NI, NI)   |
| b. del $R_1$                                     | (del $R_2$ , DR)  | (del $R_2$ , DR)   | $(R_2 \xrightarrow{-pt} R_2^l, \text{DR})$   |
| c. $R_1 \xrightarrow{+pt} R_1^l$                 | (NI, NI)  | (NI, DR)  <br>$(R_2 \xrightarrow{+pt} R_2^l, \text{NI})$   | (NI, DR)   |
| d. $R_1 \xrightarrow{-pt} R_1^l$                 | (NI, NI)<br>(NI, DR)*  <br>$(R_2 \xrightarrow{-pt} R_2^l, \text{NI})$  <br>(del $R_2$ , DR) | $(R_2 \xrightarrow{-pt} R_2^l, \text{NI})$  <br>$(R_2 \xrightarrow{-pt} R_2^l, \text{DR})^*$                                 | $(R_2 \xrightarrow{-pt} R_2^l, \text{NI})$   |
| e. $R_1 \xrightarrow{pt \rightarrow pt^l} R_1^l$ | (NI, NI)  <br>$(R_2 \xrightarrow{pt \rightarrow pt^l} R_2^l, \text{NI})$                    | $(R_2 \xrightarrow{pt \rightarrow pt^l} R_2^l, \text{NI})$  <br>$(R_2 \xrightarrow{pt \rightarrow pt^l} R_2^l, \text{DR})^*$ | $(R_2 \xrightarrow{pt \rightarrow pt^l} R_2^l, \text{N})$<br>$(R_2 \xrightarrow{pt \rightarrow pt^l} R_2^l, \text{DR})$              |
| f. $R_1 \xrightarrow{+ct} R_1^l$                 | (NI, NI)  <br>(NI, DR)  <br>$(R_2 \xrightarrow{+ct} R_2^l, \text{NI})$                      | (NI, NI)   | (NI, NI)   |
| g. $R_1 \xrightarrow{-ct} R_1^l$                 | (NI, NI)  <br>$(R_2 \xrightarrow{-ct} R_2^l, \text{NI})$                                    | (NI, NI)  <br>(NI, DR)*  <br>$(R_2 \xrightarrow{-ct} R_2^l, \text{NI})$  <br>$(R_2 \xrightarrow{-ct} R_2^l, \text{DR})^*$    | (NI, NI)  <br>(NI, DR)  <br>$(R_2 \xrightarrow{-ct} R_2^l, \text{NI})$  <br>$(R_2 \xrightarrow{-ct} R_2^l, \text{DR})$               |
| h. $R_1 \xrightarrow{ct \rightarrow ct^l} R_1^l$ | (NI, NI)  <br>$(R_2 \xrightarrow{ct \rightarrow ct^l} R_2^l, \text{NI})$                    | (NI, NI)  <br>$(R_2 \xrightarrow{ct \rightarrow ct^l} R_2^l, \text{NI})$   | (NI, NI)  <br>$(R_2 \xrightarrow{ct \rightarrow ct^l} R_2^l, \text{NI})$   |
| i. del $R_2$                                     | $(R_1 \xrightarrow{-pt} R_1^l, \text{DR})$  | (del $R_1$ , DR)   | (del $R_1$ , DR)   |
| j. $R_2 \xrightarrow{+pt} R_2^l$                 | (NI, DR)  <br>$(R_1 \xrightarrow{+pt} R_1^l, \text{NI})$                                    | (NI, DR)**  <br>$(R_1 \xrightarrow{+pt} R_1^l, \text{NI})$   | (NI, NI)   |
| k. $R_2 \xrightarrow{-pt} R_2^l$                 | $(R_1 \xrightarrow{-pt} R_1^l, \text{NI})$  | $(R_1 \xrightarrow{-pt} R_1^l, \text{NI})$  <br>$(R_1 \xrightarrow{-pt} R_1^l, \text{DR})^*$                                 | (NI, NI)  <br>(NI, DR)**  <br>$(R_1 \xrightarrow{-pt} R_1^l, \text{NI})$  <br>(del $R_1$ , DR)                                       |
| l. $R_2 \xrightarrow{pt \rightarrow pt^l} R_2^l$ | $(R_1 \xrightarrow{pt \rightarrow pt^l} R_1^l, \text{NI})$                                  | $(R_1 \xrightarrow{pt \rightarrow pt^l} R_1^l, \text{NI})$  <br>$(R_1 \xrightarrow{pt \rightarrow pt^l} R_1^l, \text{DR})^*$ | (NI, NI)<br>$(R_1 \xrightarrow{pt \rightarrow pt^l} R_1^l, \text{NI})$<br>$(R_1 \xrightarrow{pt \rightarrow pt^l} R_1^l, \text{DR})$ |
| m. $R_2 \xrightarrow{+ct} R_2^l$                 | (NI, DR)**  <br>$(R_1 \xrightarrow{+ct} R_1^l, \text{NI})$                                  | (NI, DR)  <br>$(R_1 \xrightarrow{+ct} R_1^l, \text{NI})$   | (NI, NI)  <br>$(R_1 \xrightarrow{+ct} R_1^l, \text{NI})$   |
| n. $R_2 \xrightarrow{-ct} R_2^l$                 | $(R_1 \xrightarrow{-ct} R_1^l, \text{NI})$  | $(R_1 \xrightarrow{-ct} R_1^l, \text{NI})$  <br>$(R_1 \xrightarrow{-ct} R_1^l, \text{DR})^{**}$                              | (NI, NI)  <br>$(R_1 \xrightarrow{-ct} R_1^l, \text{NI})$  <br>$(R_1 \xrightarrow{-ct} R_1^l, \text{DR})$                             |
| o. $R_2 \xrightarrow{ct \rightarrow ct^l} R_2^l$ | $(R_1 \xrightarrow{ct \rightarrow ct^l} R_1^l, \text{NI})$                                  | $(R_1 \xrightarrow{ct \rightarrow ct^l} R_1^l, \text{NI})$   | (NI, NI)  <br>$(R_1 \xrightarrow{ct \rightarrow ct^l} R_1^l, \text{NI})$   |
| p. Del Relation                                  | (NI, NI)  | (NI, NI)   | (NI, NI)   |

Table 5.1: Change impact alternatives table (part 1). \* indicates that both related requirements become equivalent, \*\* indicates possibility of a different type of relationship

| Change   | Case 4                                       | Case 5                              |
|--|--|-------------------------------------|
|  | $R_1 \xrightarrow{\text{req}} R_2$           | $R_1 \xrightarrow{\text{conf}} R_2$ |
| a. add $R_x$   | (NI, NI)                                     | (NI, NI)                            |
| b. del $R_1$   | (NI, DR)  <br>(del $R_2$ , DR)               | (NI, Dr)                            |
| c. $R_1 \xrightarrow{+pt} R_1^l$                     | (NI, NI)                                     | (NI, NI)                            |
| d. $R_1 \xrightarrow{-pt} R_1^l$                     | (NI, NI)  <br>(NI, DR)  <br>(del $R_2$ , DR) | (NI, NI)  <br>(NI, DR)              |
| e. $R_1 \xrightarrow{pt \leftrightarrow pt^l} R_1^l$ | (NI, NI)  <br>(NI, DR)  <br>(del $R_2$ , DR) | (NI, NI)  <br>(NI, DR)              |
| f. $R_1 \xrightarrow{+ct} R_1^l$                     | (NI, NI)                                     | (NI, NI)                            |
| g. $R_1 \xrightarrow{-ct} R_1^l$                     | (NI, NI)  <br>(NI, DR)  <br>(del $R_2$ , DR) | (NI, NI)  <br>(NI, DR)              |
| h. $R_1 \xrightarrow{ct \leftrightarrow ct^l} R_1^l$ | (NI, NI)  <br>(NI, DR)  <br>(del $R_2$ , DR) | (NI, NI)  <br>(NI, DR)              |
| i. del $R_2$   | (NI, DR)  <br>(del $R_1$ , DR)               | (NI, DR)                            |
| j. $R_2 \xrightarrow{+pt} R_2^l$ (NI, NI)            | (NI, NI)                                     |                                     |
| k. $R_2 \xrightarrow{-pt} R_2^l$                     | (NI, NI)  <br>(NI, DR)  <br>(del $R_1$ , DR) | (NI, NI)  <br>(NI, DR)              |
| l. $R_2 \xrightarrow{pt \leftrightarrow pt^l} R_2^l$ | (NI, NI)  <br>(NI, DR)  <br>(del $R_1$ , DR) | (NI, NI)  <br>(NI, DR)              |
| m. $R_2 \xrightarrow{+ct} R_2^l$                     | (NI, NI)                                     | (NI, NI)                            |
| n. $R_2 \xrightarrow{-ct} R_2^l$                     | (NI, NI)  <br>(NI, DR)  <br>(del $R_1$ , DR) | (NI, NI)  <br>(NI, DR)              |
| o. $R_2 \xrightarrow{ct \leftrightarrow ct^l} R_2^l$ | (NI, NI)  <br>(NI, DR)  <br>(del $R_1$ , DR) | (NI, NI)  <br>(NI, DR)              |
| p. Del Relation                                      | (NI,NI)                                      | (NI,NI)                             |

Table 5.2: Change impact alternatives table (part 2). \* indicates that both related requirements become equivalent, \*\* indicates possibility of a different type of relationship

Consider the following derivation for the domain change of adding a property to  $R_1$  while  $R_1$  *contains*  $R_2$ , which provides only a single propagation.

**Example: derivation for Change c. Case 1:**  $R_1 \xrightarrow{+pt} R_1^l \times R_1$  *contains*  $R_2$

- Let  $RM$  be such that it captures requirements  $R_1$  and  $R_2$ .
- Let  $R_1 = \langle P_1, S_1 \rangle$  and  $R_2 = \langle P_2, S_2 \rangle$  such that  $R_1$  *contains*  $R_2$ , thus  $P_1 = P_2 \wedge P_{other}$ .
- Thus  $RM = \langle P_{RM}, S_{RM} \rangle$ , with  $P_{RM} = P_1 \wedge P_2$  and  $S_{RM} = S_1 \cap S_2$ .
- Let the domain change be adding a property to  $R_1$ , thus  $R_1 \xrightarrow{+pt} R_1^l$ , with  $P_1^l = P_2 \wedge P_{other} \wedge P_{pt}$ .
- Then  $RM \neq RM^l$  holds, for  $RM^l = \langle P_{RM}^l, S_{RM}^l \rangle$ , with  $P_{RM}^l = P_1^l \wedge P_2$ .
- The requirements relation  $R_1^l$  *contains*  $R_2$  is still valid, thus there is no choice to propagate the change.
- Thus the impact alternative is no impact on related requirement, no impact on relation; (no impact, no impact).

Consider the following derivation for the domain change of adding a constraint to  $R_2$  while  $R_1$  *refines*  $R_2$  that has propagation alternatives.

**Example: derivation for Change m. Case 2:**  $R_2 \xrightarrow{+ct} R_2^l \times R_1$  *refines*  $R_2$

- Let  $RM$  be such that it captures requirements  $R_1$  and  $R_2$ .
- Let  $R_1 = \langle P_1, S_1 \rangle$  and  $R_2 = \langle P_2, S_2 \rangle$  such that  $R_1$  *refines*  $R_2$
- Thus  $P_1 = (p'_1 \cdots p'_i) \wedge (q_1 \cdots q_j)$  and  $P_2 = (p_1 \cdots p_i) \wedge (q_1 \cdots q_j)$  with  $i \geq 1, j \geq 0$ , such that  $p'_i \rightarrow p_i$
- Thus  $RM = \langle P_{RM}, S_{RM} \rangle$ , with  $P_{RM} = P_1 \wedge P_2$  and  $S_{RM} = S_1 \cap S_2$ .
- Let the domain change be adding a constraint to a property to  $R_2$ , thus  $R_2 \xrightarrow{+ct} R_2^l$ .
- Then the added constraint is either added to the property described by  $(p_1 \cdots p_i)$  or  $(q_1 \cdots q_j)$ :
  - Constraint added to  $(p_1 \cdots p_i)$ :  $P_2^l = (p'_1 \cdots p'_i) \wedge (q_1 \cdots q_j)$
  - Constraint added to  $(q_1 \cdots q_j)$ :  $P_2^l = (p_1 \cdots p_i) \wedge (q'_1 \cdots q'_j)$
- In either case,  $RM \neq RM^l$  holds, for  $RM^l = \langle P_{RM}^l, S_{RM}^l \rangle$
- But relation  $R_1$  *refines*  $R_2^l$  is no longer valid because
  - if constraint is added to  $(p_1 \cdots p_i)$  then  $p'_i \rightarrow p'_i$  does not hold
  - if constraint is added to  $(q_1 \cdots q_j)$  then  $q_i \rightarrow q'_i$  does not hold

- Thus the requirement engineer is posed with the choice to either propagate the change, to maintain the *refines* relation, such that the impact to  $R_1$  is add constraint. Or he can choose to delete relation.
- Thus  $(R_1 \xrightarrow{+ct} R_1^l, \text{no impact})$  or  $(\text{no impact}, \text{delete relation})$ .

### 5.2.5 Explanation and limitation of the change impact table

As depicted in Table 5.1 some impact possibilities are indicated with one or two asterisks. Given a single requirement change and after applying the change the possibility exists that two requirements become equivalent (indicated by \*). Similarly, the alternatives marked with a double asterisk (\*\*) indicate cases where an other relation may be valid. In none of the cases for the *conflicts* relation, an impact is determined for the related requirement. The only alternatives are to reflect either no impact at all, or a deletion of relation.

From the impact alternatives table, depicted in Table 5.1 the following observations are made, which are explained in more detail in the following paragraphs:

1. Adding a requirement to the model causes no impact
2. Adding a property to a container property can not propagate to the contained property
3. There are possibilities where two requirements become equivalent (indicated by \*)
4. There are possibilities where there are emerging requirements relations (indicated by \*\*)
5. Cases that concern the ‘conflicts’ relation never lead to propagation
6. The only propagated impact for the cases concerning ‘requires’ is ‘delete requirement’

#### ‘Add requirement’ causes no impact

The requirements relations existing in the model are used to determine and propagate impact. When adding a new requirement to the model, there are no relations connected to the requirement. These need to be manually added to the model by the requirements engineer first. In the situation where the added requirement captures properties that contradict other requirements, conflicts relations can be used to indicate this. Consequently, changes can be made to the model to resolve these conflicts relations. These changes in turn may be model changes of a type other than ‘add requirement’, and in turn propagate change using the newly added requirements relations.

Automatic trace detection is out of scope of this thesis.

#### Propagate only the needed change

Propagation of change is determined by the rules as presented in subsection 5.2.2. As a result, a change is propagated as long as it is determined by the

requirements relation that  $RM \neq RM^l$  does not hold. Alternatively, when this holds, a change may optionally be propagated if it can be determined that without propagation the relation does no longer hold. If this can not be determined, propagation of change is not considered a ‘needed change’ and can not be derived, and are model changes from refactoring.

Due to propagating *only* the needed change, a clear distinction is made between impact propagation and refactoring performed to the requirements model. If a relation is ensured to remain valid after the change, impact on the related requirement can not be determined.

The requirements engineer is responsible for refactoring the model to include changes to requirements that are not considered *needed* or cause existing relations to become invalid.

### Requirements becoming equivalent

The change impact alternatives table indicates a few cases where an alternative indicates where both requirements become equivalent (indicated by \*). As a result, the original requirement relation is no longer valid. There is no change classification nor relation to indicate the equivalence of two requirements. For determining impact propagation, only the change type and the requirements relation are considered. The explicit formula of properties that the requirement captures are not considered. Using this information actual equivalence can not be determined, and *only* possibility of the relation becoming invalid can be given.

Consider the following example:

**Example:**  $R_1 \xrightarrow{-pt} R_1^l \times R_1 \xrightarrow{\text{contains}} R_2$  (Change d. Case 1)

- Let  $RM$  be such that it captures requirements  $R_1$  and  $R_2$
- Let  $R_1 = \langle P_1, S_1 \rangle$  and  $R_2 = \langle P_2, S_2 \rangle$  such that  $R_1$  *contains*  $R_2$ , thus  $P_1 = P_2 \wedge P_{other}$ .
- Then  $RM = \langle P_{RM}, S_{RM} \rangle$ , such that  $P_{RM} = P_2 \wedge P_{other}$
- Let deleting a property from  $R_1$  be  $R_1 \xrightarrow{-pt} R_1^l$ , where  $R_1^l = \langle P_1^l, S_1^l \rangle$  is such that  $P_1^l = P_2$  and  $S_1^l = S_2$
- Then  $R_1^l = R_2$
- Thus  $RM^l = \langle P_{RM}^l, S_{RM}^l \rangle$ , such that  $P_{RM}^l = P_2$ , and  $RM \neq RM^l$  holds
- However,  $S_1^l \subset S_2$  does not hold and thus the *contains* relation is no longer valid.
- As a result, the relation and should be removed.

Although both requirements are equivalent after the change, the only indicated impact is removing the relation. Changing the model to reflect that both requirements are the same is considered refactoring and should be performed by the requirements engineer manually.



### Newly emerging relations

Similar to the cells marked with an asterisk, there are cases where the relation is removed while another relation may be valid (indicated by \*\*). The indicated impact however is delete relation. The same rationale applies here, that actual properties captured by the requirements are not taken into account, only the change type and relation. Emerging relations can therefore not be determined automatically and is left to requirements engineer.

Consider the example:

**Example:**  $R_2 \xrightarrow{+pt} R_2^l \times R_1 \xrightarrow{\text{refines}} R_2$  (Change j. Case 2)

- Let  $RM$  be such that it captures requirements  $R_1$  and  $R_2$
- Let  $R_1 = \langle P_1, S_1 \rangle$  and  $R_2 = \langle P_2, S_2 \rangle$  such that  $R_1$  *refines*  $R_2$
- Let  $P_1 = (p'_1 \cdots p'_i)$ ,  $P_2 = (p_1 \cdots p'_i)$  with  $i \geq 1$
- Let the added property to  $R_2$  be described as  $(q_1 \cdots q_j)$ , with  $j \geq 1$
- Then  $R_2^l = \langle P_2^l, S_2^l \rangle$  with  $P_2^l = (p_1 \cdots p_i) \wedge (q_1 \cdots q_j)$ ,  $i, j \geq q$
- The *refines* relation does no longer hold, and should be removed
- As a result the relation is deleted.
- By formalization of the partially *refines* relation  $R_1$  *partially refines*  $R_2^l$  holds
- This emerging relation is not indicated by the impact.

While the given relation is deleted, a different relation is valid. Maintaining the relationship between the requirements without the propagation of change is also considered refactoring and is not covered by domain change propagation.

The ‘delete relation’ *only* indicates that the relation is no longer valid, but does not give an indication whether or not an other relation holds for requirements after applying the change. The requirements engineer should be aware of these situations and refactor the model accordingly.

### No impact propagation over conflicts relation

None of the impact alternatives for cases using the *conflicts* relation indicate an impact for the requirement. This is due to the *conflicts* relation being only extensionally defined as the disjunction of sets of systems satisfying the individual requirement. There is no intentionally defined relation describing how the properties captures by requirements are related. It can therefore not be determined if there is a *direct* impact on the related requirement over the conflicts relation.

Consider the following example:

**Example:**  $R_1 \xrightarrow{-pt} R_1^l \times R_1 \xrightarrow{\text{conflicts}} R_2$  (**Change d. Case 5**)

- Let  $RM$  be such that it captures requirements  $R_1$  and  $R_2$
- Let  $R_1 = \langle P_1, S_1 \rangle$  and  $R_2 = \langle P_2, S_2 \rangle$  such that  $R_1$  *conflicts*  $R_2$
- Let  $P_1 = (p_1 \cdots p_i) \wedge (q_1 \cdots q_j)$ ,  $i, j \geq 1$
- Let  $P_2 = (p_1 \cdots p_i) \wedge (r_1 \cdots r_k)$ ,  $i, k \geq 1$
- Then  $P_1$  and  $P_2$  capture a property described by  $(p_1 \cdots p_i)$ ,  $i \geq 1$  that is captured by both requirements
- Let the deleted property be  $(p_1 \cdots p_i)$ ,  $i \geq 1$
- Then by removing this property, the impact alternatives listed are (no impact, no impact) | (no impact, delete relation).
- Thus after deleting the property, it is still captured by  $R_2$

Although deleting property  $pt$  from  $R_1$  is considered a domain change, after applying the change, and determining (either alternative) impact, property  $pt$  is still captured by the model. This is inconsistent with the formalization of domain change, unless the change of removing  $pt$  is propagated to  $R_2$  following an alternative path.

Because the conflicts relation is *only* extensionally defined, and the set of systems satisfying both requirements is empty, the change can not be related to either a set of systems that is captured by both requirements, nor a property captured by the related requirement  $R_2$  (due to lack of intentional definition of conflicts relation). The only way to delete property  $p$  from  $R_2$  is through propagation over a shared requirement  $R_x$  which captures property  $p$  and is related to both  $R_1$  and  $R_2$  through a contains relation, such that  $R_1 \xrightarrow{\text{contains}} R_x$  and  $R_2 \xrightarrow{\text{contains}} R_x$ . The requirements engineer should be aware that conflicting requirements that also share properties, should have this property modeled by a third requirement explicitly. Making such modeling decisions is the only way to ensure that  $P_{RM} \neq P_{RM}^l$ , after propagating the change.

### Only one impact type over requires relation

Similarly to the conflicts relation, the *requires* relation is also only extensionally defined. The difference between the *conflicts* and *requires* relation is, that the *requires* relation indicates that the sets of systems satisfying the requirements are related through a subset relation, rather than disjunction. This allows for course change propagation. Without intentional definitions, the exact type of propagation of change can not be determined, and thus the only propagation of impact that can be determined is deleting the related requirement to ensure that  $RM \neq RM'$  holds. The requirements engineer should be aware of this coarseness of impact over requires relations.

### 5.2.6 Change Impact Prediction

#### Prediction categories

The formalism of the requirements model and the formalism for domain changes allows the distinction between actual propagation, potential propagation, and no propagation to related requirements. This leads to the following three categories of possibility of propagation:

1. **Ensured propagation:** The relation leading from the impacted requirement to the related requirement *ensures* that  $\neg(RM \neq RM^l)$  if the related requirement is not changed. As a result the related requirement *must* be changed.
2. **Possible propagation:** The requirements relation leading from the impacted requirement to the related requirement leads to a combination of following possibilities, which requires further investigation by the requirements engineer:
  - a.  $RM \neq RM^l$  is ensured and the relation is ensured to hold, and thus no impact on the related requirement.
  - b.  $RM \neq RM^l$  is ensured, but the relation is not ensured to hold. The requirement engineer may propagate the change or delete relation.
  - c.  $RM \neq RM^l$  is not ensured if the related requirement is not changed, and thus the related requirement is impacted.
3. **No propagation:** The relation leading from the impacted requirement to the related requirement ensures that  $RM \neq RM^l$ . There is no possibility of propagating the change. The relation may become invalid however.

These categories allow change impact prediction, given a change and a requirement relation. The result is an indication of the degree of propagation. The possibility of propagation is derived from the change impact alternatives table where each of the possible changes to requirements elements as mentioned in section 4.3 with cases which contain the five requirements relation *contains*, *refines*, *requires*, *conflicts* and *partially refines* are combined.

Using the categorization for possibility of propagation the propagation possibility for each combination of requirements relation and requirements model change are derived. The results are depicted in Table 5.3. This table describes the likelihood of impacting the related requirement.

This table can be used to cover the ‘base’ scenario of a single elementary change that is introduced to the requirements model. For this, all requirements relations fanning out from the impacted requirements are considered, together with the classification of change.

Given a change type on a certain requirement, reachability analysis is performed. The degree of propagation possibility is indicated by the change impact prediction rules. The analysis results in an overview of reachable requirements with an indication of the possibility of impact. E.g. the prediction yields ‘ensured impact’, ‘possible impact’ or ‘no impact’ for each requirement.

|    | Change                                    | Case 1                              | Case 2                             | Case 3                               | Case 4                             | Case 5                              |
|----|---|-------------------------------------|------------------------------------|--------------------------------------|------------------------------------|-------------------------------------|
|    |   | $R_1 \xrightarrow{\text{cont}} R_2$ | $R_1 \xrightarrow{\text{ref}} R_2$ | $R_1 \xrightarrow{\text{p.ref}} R_2$ | $R_1 \xrightarrow{\text{req}} R_2$ | $R_1 \xrightarrow{\text{conf}} R_2$ |
| a. | add $R_x$                                 | no                                  | no                                 | no                                   | no                                 | no                                  |
| b. | del $R_1$                                 | yes                                 | yes                                | yes                                  | maybe                              | no                                  |
| c. | $R_1 \xrightarrow{+pt} R_1^l$             | no                                  | maybe                              | no                                   | no                                 | no                                  |
| d. | $R_1 \xrightarrow{-pt} R_1^l$             | maybe                               | yes                                | yes                                  | maybe                              | no                                  |
| e. | $R_1 \xrightarrow{pt \mapsto pt^l} R_1^l$ | maybe                               | yes                                | yes                                  | maybe                              | no                                  |
| f. | $R_1 \xrightarrow{+ct} R_1^l$             | maybe                               | no                                 | no                                   | no                                 | no                                  |
| g. | $R_1 \xrightarrow{-ct} R_1^l$             | maybe                               | maybe                              | maybe                                | maybe                              | no                                  |
| h. | $R_1 \xrightarrow{ct \mapsto ct^l} R_1^l$ | maybe                               | maybe                              | maybe                                | maybe                              | no                                  |
| i. | del $R_2$                                 | yes                                 | yes                                | yes                                  | maybe                              | no                                  |
| j. | $R_2 \xrightarrow{+pt} R_2^l$             | maybe                               | maybe                              | no                                   | no                                 | no                                  |
| k. | $R_2 \xrightarrow{-pt} R_2^l$             | yes                                 | yes                                | maybe                                | maybe                              | no                                  |
| l. | $R_2 \xrightarrow{pt \mapsto pt^l} R_2^l$ | yes                                 | yes                                | maybe                                | maybe                              | no                                  |
| m. | $R_2 \xrightarrow{+ct} R_2^l$             | maybe                               | maybe                              | maybe                                | no                                 | no                                  |
| n. | $R_2 \xrightarrow{-ct} R_2^l$             | yes                                 | yes                                | maybe                                | maybe                              | no                                  |
| o. | $R_2 \xrightarrow{ct \mapsto ct^l} R_2^l$ | yes                                 | yes                                | maybe                                | maybe                              | no                                  |
| p. | Del Relation                              | no                                  | no                                 | no                                   | no                                 | no                                  |

Table 5.3: Change prediction table; ensured propagation (yes), possible propagation (maybe), and no propagation (no)

### Change Type Prediction

Additional to providing impact prediction, the classification of change can be indicated as well. By repeatedly and exhaustively applying the change impact alternatives table for the initial change, all possible choices for change propagation are determined. This can be regarded as constructing a decision tree, where the initial change is the starting node in the tree. Outgoing edges from the initial node indicate each possible propagation alternative over each relation connected to the initially changed requirement.

For each possible alternative a new decision tree is constructed. This way all reachable requirement over the requirements relations are reached, with all possible propagation alternatives. This process can be regarded as exhaustively constructing a decision tree for all reachable requirements from the initially impacted requirement.

The set of all possible propagation paths and their corresponding impacts can be used to provide both the indication of the change type when an impact is determined, as well as the propagation path from the initially changed requirement to the impacted requirement.

Consider the following example for generating a decision tree:

**Example: decision tree**

Consider the requirements model as depicted in Figure 5.1, with ‘delete property’ as initial change on  $R_2$ .

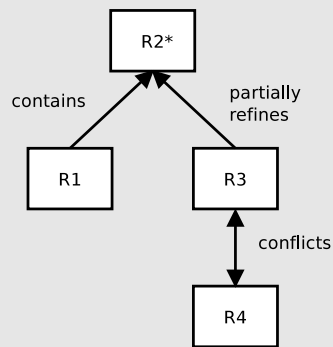


Figure 5.1: Example requirements model with  $R_2$  in SIS

Reachability analysis, based on a depth-first search yields the two paths and their reached requirements:

1.  $R_2, R_3, R_4$
2.  $R_2, R_1$

For both paths, the decision trees are constructed, depicted in Figure 5.2 and Figure 5.3.

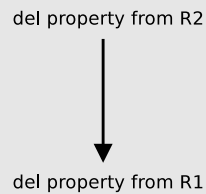


Figure 5.2: Generated decision tree for  $R_2, R_1$

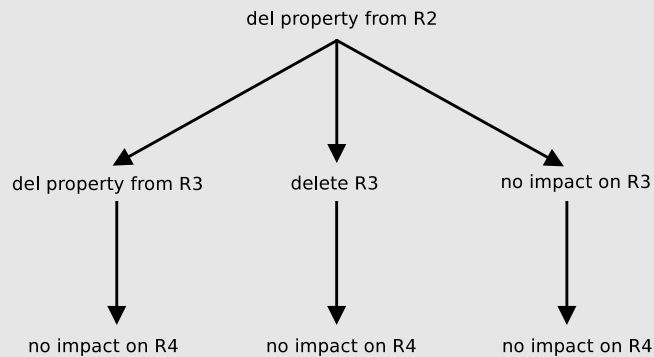


Figure 5.3: Generated decision tree for  $R_2, R_3, R_4$

The result of the change impact prediction is depicted in Table 6.4.

| Requirement | Impacted | Impact types                                      |
|-------------|----------|---|
| $R_1$       | yes      | delete property                                   |
| $R_2$       | yes      | delete property                                   |
| $R_3$       | maybe    | no impact, delete property,<br>delete requirement |
| $R_4$       | no       | no impact   |

Table 6.4: Change impact prediction results including change types

### 5.2.7 Relation validation

Related requirements can be impacted by one and the same initial change *once*. This is a result of Bohner’s definitions of *directly* and *indirectly* impacted requirements, that an *indirectly* impacted requirement is reached by an *acyclic* path [1]. However, Bohner’s definition does not cover multiple (different) paths leading from the initially impacted requirement to an indirectly impacted requirement.

Consider the following example:

#### Example: multiple propagation paths to same requirement

Consider the example requirements model depicted in Figure 5.4, where  $R_1$  is the initially impacted requirement.

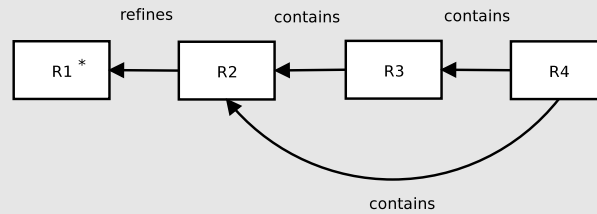


Figure 5.4: Requirements model with  $R_1$  in SIS.

Two different propagation paths exist to indirectly impact  $R_4$ :

**Path 1:** From  $R_1$  to  $R_2$ ; From  $R_2$  to  $R_3$ ; From  $R_3$  to  $R_4$

**Path 2:** From  $R_1$  to  $R_2$ ; From  $R_2$  to  $R_4$

Consider ‘Path 1’; the propagation from  $R_2$  to  $R_4$  is determined over  $R_3$ . However, the relation between  $R_2$  to  $R_4$  is not (yet) considered. A propagation from  $R_4$  back to  $R_2$  is not possible; for this would lead to an *acyclic* path. Bohner’s definition however does not state anything about the additional propagation from  $R_2$  to  $R_4$ .

In this approach a requirement reached by multiple paths from the same changed requirement can only be impacted *once* by the same initial change.

The first determined impact on a requirement from an initial change determines the impact on the requirement. Additional relations that result in an additional path to an already impacted element should be checked for validity.

### 5.3 Change consistency checking

The impact alternatives listed in Table 5.1 lists changes, following the classification of change to the requirements model. A domain change is regarded as a composition of multiple changes to the requirements model. These requirement model changes are composed of requirement changes. This is depicted in a UML diagram in Figure 5.5.

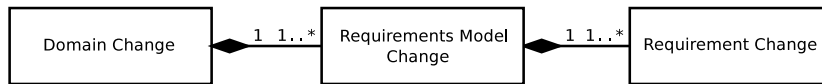


Figure 5.5: UML diagram representing structure of requirement model changes at the start of performing CIA

The simplest domain change consists of a single requirements model change, that in turn consists of one requirement change. Domain changes can lead to complex requirement model changes. These are composed of multiple affected requirements.

Additionally to the complexity of the model change, it may also be desirable to analyse the different change impact alternatives.

Thus, three dimensions of complexity for performing CIA are identified:

1. The number of unique domain changes that impact the requirements model
2. The number of requirements that are initially identified as changed
3. The number of alternative impact propagation paths that are considered

The first two dimensions may lead to situations where multiple impacts are determined on a single requirement. In these cases, a contradiction may arise. Inconsistencies resulting from multiple impacts can be indicated to be *ensured*, *possible* or *absent*. Below are listed examples for an *ensured* and *possible* inconsistency.

#### Example: ensured inconsistency

Consider two impacts to be determined for  $R$ , which lead to an *ensured inconsistency*:

1.  $R \xrightarrow{+pt} R^l$
2. *delete*  $R$

In  $R \xrightarrow{+pt} R^l$ ,  $R^l$  is formalized as:

- $R^l = (p_1 \cdots ptn) \wedge (q_1 \cdots q_m)$ , with  $n, m \geq 1$

Where  $(q_1 \cdots q_m)$  indicates the properties captured by  $R$  before the change and  $(p_1 \cdots p_n)$  represents the added property  $pt$  to  $R$ .  $R^l$  should hold after the change.

*delete*  $R$  is defined as:

- $R = (q_1 \cdots q_m)$ ,  $m \geq 1$  does not hold

This leads to an ensured contradiction because  $(q_1 \cdots q_m)$ ,  $m \geq 1$  should both hold and not hold. Thus both ‘add property’ and ‘delete requirement’ impacts on the same requirement leads to an ensured inconsistency.

### Example: possible inconsistency

Consider two impacts to be determined for  $R$ , which lead to a *possible inconsistency*:

1.  $R \xrightarrow{pt \rightarrow pt^l} R^l$
2.  $R \xrightarrow{-pt} R^l$

In  $R \xrightarrow{pt \rightarrow pt^l} R^l$ ,  $R$  and  $R^l$  are formalized as:

- $R = (p_1 \cdots p_n) \wedge (q_1 \cdots q_w)$ , with  $n \geq 1, m \geq 0$
- $R^l = (t_1 \cdots t_z) \wedge (q_1 \cdots q_w)$ , with  $z \geq 1, m \geq 0$

Where  $(p_1 \cdots p_n)$  indicates the properties captured by  $pt$  and  $(t_1 \cdots t_z)$  indicates the properties captured by  $pt^l$ .

$R \xrightarrow{-pt} R^l$  is formalized as:

- $R = (p_1 \cdots p_i) \wedge (q_1 \cdots q_j)$ , with  $i, j \geq 1$
- $R^l = (q_1 \cdots q_j)$ , with  $j \geq 1$

Where  $(p_1 \cdots p_i)$  indicates the properties captured by  $pt$  that are removed from the requirement. Thus after deleting the property from  $R$ ,  $(p_1 \cdots p_i)$ ,  $i \geq 1$  should no longer hold.

Then either situation occurs:

1. The disjunction of literals in CNF described by  $(p_1 \cdots p_i)$ ,  $i \geq 1$  are not captured by the replaced property described by  $(t_1 \cdots t_z)$ ,  $z \geq 1$ , e.g.  $\forall p_x, x \in i : p_x \neq t_y$ , with  $y \in \{1, \dots, z\}$
2. The disjunction of literals in CNF described by  $(p_1 \cdots p_i)$ ,  $i \geq 1$  captures at least one literal such that  $\exists p_x, x \in i : p_x = t_y$ , with  $y \in \{1, \dots, z\}$

In the first case, no contradiction is determined, and both changes can be performed on the requirement. In the second case, a contradiction arises because a literal is still captured by  $R^l$  that should no longer hold. Thus a requirement both impacted by ‘change property’ and ‘delete property’ *can* be inconsistent.



The Table 5.5 gives the contradicting changes based on semantics of domain changes and change types in change classification.

|    | Change                                    | a.  | b.  | c.    | d.    | e.    | f.    | g.    | h. |
|----|---|-----|-----|-------|-------|-------|-------|-------|----|
| a. | del $R$                                   | no  | yes | no    | yes   | yes   | no    | yes   | no |
| b. | $R \stackrel{+pt}{\mapsto} R'$            | yes | no  | no    | no    | no    | no    | no    | no |
| c. | $R \stackrel{-pt}{\mapsto} R'$            | no  | no  | no    | maybe | maybe | no    | maybe | no |
| d. | $R \stackrel{pt \mapsto pt'}{\mapsto} R'$ | yes | no  | maybe | maybe | maybe | maybe | maybe | no |
| e. | $R \stackrel{+ct}{\mapsto} R'$            | yes | no  | maybe | maybe | no    | maybe | maybe | no |
| f. | $R \stackrel{-ct}{\mapsto} R'$            | no  | no  | no    | maybe | maybe | no    | maybe | no |
| g. | $R \stackrel{ct \mapsto ct'}{\mapsto} R'$ | yes | no  | maybe | maybe | maybe | maybe | maybe | no |
| h. | no impact                                 | no  | no  | no    | no    | no    | no    | no    | no |

Table 5.5: Table indicating the possible inconsistencies when a requirement is affected by multiple impacts

According to the table, two changes for the same requirement are classifiable by three degrees of contradiction:

**Yes:** It is ensured that both impacts lead to a contradiction

**Maybe:** Impacts *may* cause a contradiction, and this should be investigated

**No:** The absence of contradiction is ensured

## 5.4 Discussion of the approach

In this section, the various parts of the approach are discussed.

### 5.4.1 Formalization in FOL

FOL is used to formalize the semantics of classification of changes, change rationale and propagation of changes. There are limitations to the expressiveness of FOL. Permissible or obligatory properties, indicating a degree of possibility can not be expressed using FOL. There are other formalizations of requirements which can be used to express these properties, for example modal and deontic logic[31][32].

The formalization in FOL allows the expression of changes in commonly occurring requirements descriptions, including for example real-time or performance requirements. The expressiveness of FOL is considered sufficient for inferencing of requirements relations and consistency checking of the formalized requirements model. In this work, the expressiveness of FOL is considered sufficient to use for semantics for classification of change and rationale of change.

### 5.4.2 Mapping textual requirements to requirement model

Modeling of the requirements and requirements relations is carried out by the requirements engineer. How and which textual requirements should be represented in the requirements model is not determined by the approach. The approach is agnostic about differences between functional and non-functional requirements. Thus the requirement engineer is free to construct his model and determine relations as he sees fit.

Construction of the model abstracts from the actual formalization of the requirement. The decomposition into properties and constraints is not explicitly captured by the model. The requirements engineer can be guided by the informal definitions of semantics of the relations and can be guided by tutorials[25].

The benefit is that although the semantics of requirements relations needs to be known, knowledge to formalize the requirements is not required. The learning curve is therefore less steep and time consuming than when explicit formalization of requirements is needed.

By abstracting from the explicit mapping of textual requirements requirements to properties and their predicates certain capabilities are lost, such as:

- Automatic detection of equivalent requirements due to a change.
- Automatic trace generation resulting from changed/added requirements

The benefits of the underlying semantics can still be used. By using the tool support for inferencing of requirements relations and checking model consistency, that allow for improving the model[2].

### 5.4.3 Classification and propagation of changes

The structure of the formalized requirement is derived from Wasson's primitives[3] to determine the different types of changes that can be made to the formalized requirement. By having a one-to-one mapping from the Wasson's primitives onto the formalized requirement, knowledge of the textual domain can be used in the requirements model.

By defining the classification of change in terms closely related to terms that are known in the textual domain, an abstraction is made from the specific changes in the FOL formalization. This way, the requirements engineer can perform CIA on textual requirements and still use well-defined requirements relations. This implies a less steep learning curve, for the requirements engineer can be guided by tutorials.

As mentioned before, the model does not capture explicit mapping to FOL. The change classification does not explicitly indicate to which property or properties the change in the requirement is applied. As a result some loss of information occurs. Specific change propagations such as those where requirements become equivalent or where new relations emerge can not be automatically detected. This means that to keep track of what *actually* has to be changed in the requirements, the requirements engineer has to keep track of the changes in the textual requirement.

The propagation of change is driven by domain change. Refactoring does not imply *needed* changes, for the systems described by the sum of all requirements remain the same. From the working definition of impact as 'the needed

change’ and the requirements relations to and from the change requirement, change propagation is determined. By using the domain change rather than refactoring, the ‘needed’ change is determined. This provides clear semantics for propagation.

The drawback of this separation of ‘domain change’ propagation and subsequent ‘refactoring’ is that for each application of changes to the model, the requirements engineer may have to refactor the model. Where these steps were previously intertwined, they are now interleaved.

#### 5.4.4 Semantics of requirements relations

Requirements relations can be categorized as ‘extensionally’ defined and ‘intentionally’ defined. Intentionally defined relations such as *contains*, *refines*, *partially refines* allows for deriving change impact propagation alternatives based on both the sets of systems satisfying the requirement *and* the properties over which the intentionally defined relations are given. This leads to more precise impact alternatives, as can be seen in the change impact alternatives Table 5.1 on page 34.

The propagated types of change for the intentionally defined relations capture all types. The propagated types of change for the extensionally defined relations *requires* and *conflicts* only capture ‘no impact’ and ‘delete requirement’.

The semantics for the ‘conflicts’ relation specifically are ill-suited for the use of change impact analysis using direct impacts, as illustrated in paragraph 5.2.5. The requirements engineer should be aware of this issue, and make according modeling choices to ensure proper (indirect) propagation of change when dealing with conflicts relations.

Requirements relations in the requirements metamodel may imply other requirements, and effectively this means that multiple relations may exist between two requirements. For example, the *contains* relation implies that the *requires* relation also holds, and thus both requirements are related through a *contains* relation and a *requires* relation. When propagating change over the requirements relation, the ‘strongest’ explicitly denoted requirements relation is taken into account, e.g. the *contains* relation.

## 5.5 Conclusion

In this chapter the semantics of change rationale and semantics for change types are used together with the formalized requirements relations to determine change propagation rules, which in turn lead to change impact alternatives (section 5.2).

The formalization can be used to determine ensured impacted, possibly impacted and unimpacted requirements using the generation of decisiontrees (subsection 5.2.6), as well as to validate requirements relations (subsection 5.2.7).

When dealing with the analysis of multiple requirement changes at the same time, the formalization of change types and change propagation allows for impact consistency checking, that can indicate the ensured, possible and absence of impact inconsistencies (section 5.3)

In section 5.4 the approach is discussed. The requirements engineer should be aware of the limitations of the approach, of which the most important limitation is the possibility of deficient results of using the ‘conflicts’ relation.

# Chapter 6

## Tool support

### 6.1 Introduction

In research prior to this thesis, proof-of-concept tool support for the metamodelling approach for requirement metamodels as proposed by Göknil et al.[2] has been developed by Veldhuis[33]. The tool ‘Tool for Requirements Inference and Consistence Checking’ (TRIC) is developed using the Eclipse Rich Client Platform (Eclipse RCP)[34]. TRIC is extended to provide support for performing CIA.

This chapter describes the requirements, architecture, design and implementation of the extension of TRIC with support for CIA. First the requirements for the extension are described in section 6.2. These requirements are used to construct the activity diagrams of performing CIA using tool support by the requirements engineer. Following the requirements, the extended architecture of TRIC with the CIA components is presented in section 6.3. In this section, added and altered components and their purposes are described. In section 6.4 the design of the added and altered components in the architecture is described. Section 6.5 describes implementation decisions that are made. Section 6.6 describes the main features and usage of the implementation. This chapter is then concluded in section 6.7.

### 6.2 Requirements for TRIC-CIA

TRIC is extended with support for performing CIA. In this section, the functional requirements for performing CIA are listed. From these requirements, the activity diagram for the intended use of TRIC is determined.

#### 6.2.1 Functional requirements

The change propagation alternatives described in Chapter 5 allows for change impact prediction and consistency checking of changes. These lead to the following functional requirements for the extension of TRIC with support for change impact analysis:

**R1 Supporting change impact analysis for domain changes.** The requirements engineer identifies domain changes. The changes are proposed

to the requirements model captured by TRIC. Subsequently, analysis on the change is performed by the requirements engineer. Changes are propagated to other requirements using the change impact alternatives determined in Chapter 5. TRIC shall support updating and removing proposed changes.

**R2 Checking change consistency.** Multiple impacts from different proposed changes may lead to (possible) inconsistencies as described in section 5.3, which results from the classification of change. TRIC shall support change consistency checking of multiple changes impacting the same requirement. TRIC shall indicate the degree of inconsistency. Additionally, TRIC shall identify causes of inconsistencies.

**R3 Providing change impact prediction.** TRIC shall provide support for prediction of the possibility of impact as well as the prediction of change type for each impact. TRIC shall provide feedback as to how the change impact prediction results are related to the propagation alternatives.

**R4 Applying changes to requirements model.** When the change impact analysis is completed, TRIC shall support applying the analyzed impacts to the requirements model. After application of the changes the requirements model is updated to reflect the changes.

**R5 Providing visual feedback.** When performing CIA, TRIC shall provide visual feedback to the requirements engineer. When inconsistencies are detected a visualization of inconsistencies should be provided. Change impact predictions shall be visually represented for the requirements engineer to inspect them.

### 6.2.2 Supported activities

The activity diagram shown in Figure 6.1 depicts the activities of performing change impact analysis, done by the requirements engineer. Due to the requirements engineer being the only actor in this activity diagram, the actor symbol is left out.

**Modeling requirements:** This activity takes the requirements document as input and produces the requirements model as output. The requirements model consists of requirements and their given and inferred relations. The definitions given in Chapter 3 are used to identify the requirements relations. The modeling process is divided into three activities: *requirements reformulating*, *trace generating* and *trace validating*. This modeling process is supported by TRIC through inferring requirements relations and requirements model consistency checking. Although the activity is not part of the actual change impact analysis, it provides the requirements model on which the CIA is performed. Additionally this activity captures refactoring.

**Interpreting change request:** This activity takes the current requirements model and the change request as input. The requirements engineer identifies domain changes that result from comparison of the change request and the current requirements model.

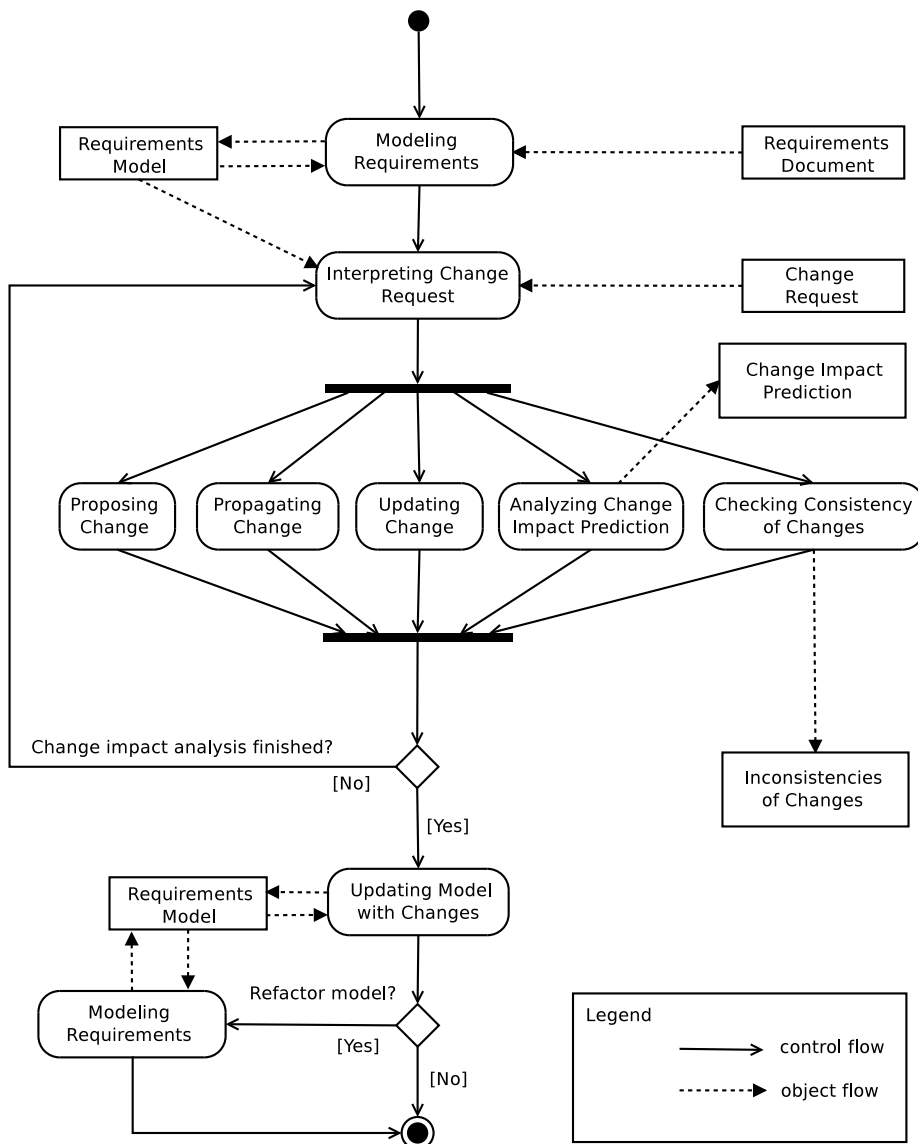


Figure 6.1: Activity diagram of performing CIA

**Proposing change:** This activity takes the requirements model and the differences between the requirements model and the requirements document as input and produces the proposed changes to the requirements model as output. The requirements engineer proposes changes to the requirements model based on the interpretation of the domain changes in stakeholder needs. This activity captures proposing one single change at once to the model. The modeling process is iterative and the requirements engineer can introduce multiple changes consecutively without propagating the proposed changes. Therefore, the performing change impact analysis process

may continue after proposing changes activity iteratively without propagating proposed changes.

**Propagating change:** This activity takes the requirements model with proposed changes as input and produces the propagated proposed changes in the requirements model as output. The activity is semi-automatic. Propagation rules are defined based on the formal definitions of the requirements relations and change types as described in subsection 5.2.4. The requirements engineer selects the propagation from the selection of possible propagations proposed by the tool. The activity denotes one propagation of a single change in the model. The modeling process is iterative and the requirements engineer may propagate multiple changes multiple times consecutively without proposing any other change.

**Updating change:** This activity takes the requirements model with proposed changes which are to be updated. The activity produces a requirements model with updated proposed changes. Propagated changes resulting from the previously unchanged change may be removed from the change impact analysis. This activity also supports the removal of proposed or propagated changes.

**Analyzing change impact prediction:** This activity takes the requirements model and a proposed change as import and gives a change impact analysis prediction for this proposed change as output. Output is the result of an automated reachability analysis that provides a prediction which indicates the possibility of impact and impact change types for requirements captured by the requirements model.

**Checking consistency of changes:** This activity takes the requirements model including the proposed and proposed propagated changes as input and gives inconsistencies between proposed changes as output. Inconsistent changes are determined, if there are any. For each inconsistency the degree of inconsistency is determined, as described in subsection 5.2.6. If inconsistencies are detected, the requirements engineer can resolve these by updating changes that lead to inconsistencies.

**Updating model with changes:** This activity takes the requirements model with proposed changes as input and produces the requirements model reflecting the proposed domain change(s) as output. The activity is semi-automated. The requirements engineer first changes the requirements in the model according to proposed changes. Then the requirements relations that may have become invalid due to the changes, are validated. Updating the model with changes can only be performed if there are no ensured inconsistencies of changes. During this activity, the set of proposed and propagated changes can no longer be changed.

The process described by the activity diagram in Figure 6.1 is iterative. Once the process has ended, the requirements engineer may start a new activity by ‘Interpreting change request’ of new domain changes to the requirements model and subsequently start a new CIA. Each individual CIA is concluded by updating the requirements model with the changes captured by the performed CIA. Considering that CIA is only provided for domain changes, refactoring the model after performing CIA is left up to the requirements engineer.



### 6.3 Architecture of TRIC

The high-level architecture for TRIC with extension for CIA support is depicted in Figure 6.2. First the existing components and their responsibilities as intended by Veldhuis[33] are listed. Then the added components and their intended responsibilities for CIA are described. Finally, the components that should be altered are listed.

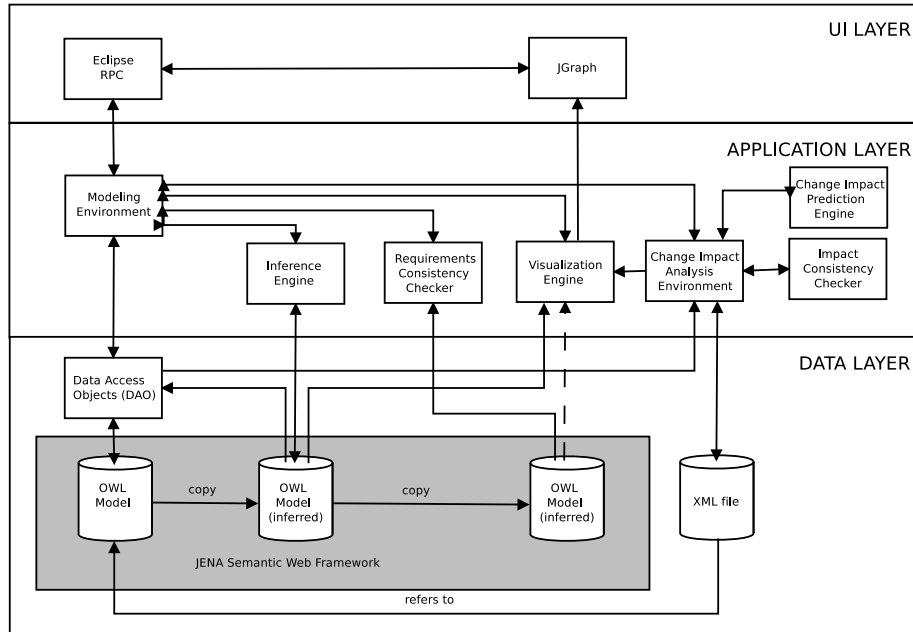


Figure 6.2: The extended layered architecture for TRIC with CIA

#### 6.3.1 Existing components

The following components are present in TRIC:

**OWL Model:** The requirement (meta)models are represented with Ontology Web Language[35] (OWL) ontology. The models are used in the JENA (FOL) reasoner to inference requirements relations and perform requirement model consistency checking. Requirements models are also stored as an OWL ontology.

**Data Access Objects:** To separate data representation and application logic, Data Access Objects (DAO) are used for internal representation of objects captured by the OWL model and are used as an intermediate layer for interaction between the Modeling Environment and the OWL model. This way the application layer can be unaware of the specific underlying structure of OWL.

**Modeling Environment:** The modeling environment is the bridge between the user interface and the internal components.

**Inference Engine:** Relations that can be derived from existing relations are determined by the inference engine. The inference engine makes use of the JENA reasoner to prepare the model and subsequently processes them so that implicitly captured relations are made explicit.

**Requirements Consistency Checker:** The consistency checker validates a requirements model with inferred relations and checks for contradictions in requirements relations.

**Visualization Engine:** This component captures the methods of constructing JGraph graphs for visual representation of views on requirements models.

**Eclipse RCP:** Eclipse Rich Client Platform is a framework for developing and deploying (client) applications. Eclipse RCP provides a platform that deals with events, controllers and UI elements. TRIC is developed as a plugin for this framework.

**JGraph:** JGraph[36] is an open-source graph visualization library, used to provide views of requirements models.

### 6.3.2 Added components

The following components extend TRIC for CIA support:

**Change Impact Analysis Environment:** Analysis of requirement changes to requirements models are performed through use of the Change Impact Analysis Environment. This component keeps track of proposed changes and propagation choices made by the requirements engineer. It provides interaction with Change Impact Prediction Engine. This component also provides the interaction with the Change Impact Consistency Checker. The component is responsible for updating the requirements model with the proposed changes once the CIA is completed.

**Change Impact Prediction Engine:** This component is responsible for automatic generation of decision trees. It determines change impact predictions and change type predictions related to a proposed change.

**Change Impact Consistency Checker:** This component checks the consistency of proposed changes. Each requirement is affected by multiple proposed changes is checked to determined if these changes are consistent.

**XML File:** The change impact analysis can be loaded from and stored to an XML file. The XML captures the current state of the change impact analysis and refers to requirements and requirements relations in the requirements model.

### 6.3.3 Altered components

The following components are altered to extend TRIC with CIA support:

**Modeling Environment:** This component is altered to interact with the Change Impact Analysis Environment. By altering the component, it also provides the user interface for the Change Impact Analysis Environment.

**Visualization Engine:** The Visualization Environment is extended to provide the visualization of change propagation paths.

## 6.4 Design

The following non-trivial design choices are made;

1. CIA is separated from the requirements model (subsection 6.4.1)
2. CIA is performed on a graph representation of the requirements model, by representing changes as annotated edges (subsection 6.4.2)
3. Only given relationships are used in CIA (subsection 6.4.3)
4. Propagation and consistency checking are not implemented using the JENA reasoner (subsection 6.4.4)
5. Depth first traversal with interleaving impact alternative annotations is used as algorithm to perform CIP (subsection 6.4.5)

Each of these design choices is described in more detail in the following subsections.

### 6.4.1 Separation of CIA and requirements model

Multiple analyses may be performed for a change. Each individual analysis should therefore be captured separately from the requirements model. Rather than capturing all changes in the requirements model, a separation is made between the requirements model and the change impact analysis. The change impact analysis is captured in a separate model.

Changes of the performed analysis are not considered final until the whole analysis process is finished. Then the changes from the analysis are considered final and are applied to the requirements model. Thus the CIA model captures the *proposed changes* and the resulting determined proposed propagated impacts.

The changes are categorized as followed:

**Proposed change:** A change for which the impact, or lack thereof, on the target requirement and relation has been determined.

**Candidate change:** A change for which the source requirement is known to be impacted, but the impact on the relation and target requirement are not yet determined.

For each proposed change, the change impact analysis model offers the possibility of adding a description for the proposed change. This allows for capturing the rationale of determined impact or the lack thereof.

### 6.4.2 Using graph representation

Because the propagation may occur in both directions over the requirements relation, the change impact model takes a representation of the requirements model as an undirected graph. Its edges have a reference to the appropriate relations in the requirements model. The vertices are references to the requirements captured in the model. The information is captured in an adjacency list which refers to the requirements described by the Modeling Environment. The graph can be further annotated to indicate additional information needed for CIA.

#### Representation of change

Changes are propagated over the requirements model relations, e.g. over the edges of the representing graph. The initially proposed change to the model is an exception to this, for it is not a change that is propagated over a relation. Instead, the initial change may be thought of to originate from outside the requirements model. Changes are represented as annotations to edges of the graph representation of the requirements model. The annotations capture the following information for propagation of change:

**Source vertex:** Reference to the requirement that is the source of the change propagation

**Target vertex:** Reference to the requirement to which the change is propagated

**Relation:** Reference to the requirements relation in the requirements model.

**Impact on requirement:** Impact on target requirement, if determined.

**Impact on relation:** Impact on requirements relation, if determined.

**Origin:** Reference to the initially proposed change of which this change is a propagated change.

**Description:** Textual representation and/or motivation of the determined impact.

The initial change is represented as an edge where the source vertex and the target vertex are the requirement that captures the initially proposed change. The relation and impact on relation annotations are not set. After an initial change is proposed, the relations of the initially impacted requirement are marked as candidate changes.

When a candidate change is investigated, the impact on the related requirement is determined. This determined information such as the description, impact on relation and impact on requirement are captured by the edge annotation. The edge is subsequently added to the list of visited edges and removed from the list of unvisited edges. Consequently, when a propagated change has been determined, newly candidate changes are determined and added to the list of candidate changes. Determining new candidate changes takes into account acyclic paths and impact on requirement.

An example CIA scenario is illustrated by the two figures, Figure 6.3 and Figure 6.4, which both depict an example requirements model. The requirements model is depicted in black, the proposed changes are depicted in red and the candidate changes are depicted in blue. In Figure 6.3 the initial change is proposed, leading to two candidate changes. In Figure 6.4 change is propagated from the initial change to a related requirement. Subsequently new candidate changes are determined.

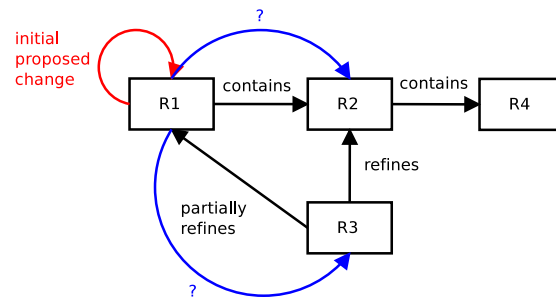


Figure 6.3: Example change impact model representation with initial proposed change in red and candidate changes in blue

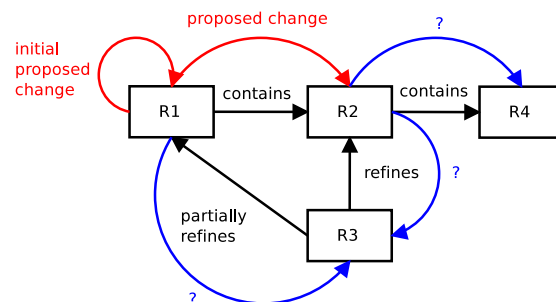


Figure 6.4: Example change impact model representation with proposed changes in red and candidate changes in blue

For each initially proposed change, a separate change impact analysis is performed. Proposed changes are regarded as annotated edges that have been visited, whereas candidate changes are regarded as annotated edges that have not been visited yet.

### Change impact matrix

A change impact matrix is created for each initially proposed change. The change impact matrix is similar to a traceability matrix, but it captures additional information of annotated candidate and proposed changes for that given initial proposed change. The change impact matrix indicates which changes are visited, and which are candidate changes.

The following figure depicts the two change impact matrices corresponding to Figure 6.3 and Figure 6.4 are given. The rows indicate the source of the propagation, columns denote the target of the propagation. Blue cells indicate candidate changes, whereas red cells indicate proposed changes. Marked cells with an  $\times$  symbol indicate a requirements relation.

|       | $R_1$    | $R_2$    | $R_3$    | $R_4$    |
|-------|----------|----------|----------|----------|
| $R_1$ |          | $\times$ | $\times$ |          |
| $R_2$ | $\times$ |          | $\times$ | $\times$ |
| $R_3$ | $\times$ | $\times$ |          |          |
| $R_4$ |          | $\times$ |          |          |

|       | $R_1$    | $R_2$    | $R_3$    | $R_4$    |
|-------|----------|----------|----------|----------|
| $R_1$ |          | $\times$ | $\times$ |          |
| $R_2$ | $\times$ |          | $\times$ | $\times$ |
| $R_3$ | $\times$ | $\times$ |          |          |
| $R_4$ |          | $\times$ |          |          |

Figure 6.5: Change impact matrices. Left matrix represents figure 6.3 before propagating the change and right matrix represents Figure 6.4 after propagating change from  $R_1$  to  $R_2$

Propagation of changes using the change impact matrix can be done until there are no more candidate changes left to determine the impact for, and the change propagation for that particular initially proposed change is done.

### Interactive decision trees

Using the change impact matrix allows for an overview of propagation of change. However, it represents the chosen propagations only. It does not allow comparison of multiple change propagation alternatives caused by the same initial change. This support is provided by performing analysis using a decision tree that captures all possible change propagation paths.

Instead of providing the exhaustive change propagation analysis, only the initial change is depicted and the requirements engineer can expand the tree interactively. This way, the different impact alternatives can be analysed separately, and multiple results of finished impact propagations can be compared to each other. After comparing the different results caused by different chosen impact alternatives, one analysis can be chosen.

The decision tree is represented using the following symbols:

**Black squares:** Represents a visited decision node

**Gray square:** Represents an unvisited decision node, indicating that the decision tree can be expanded

**Yellow square:** Represents an end node, indicating that no more decisions can be made

An illustration of a partially built decision tree is given in Figure 6.6, representing the change impact analysis after propagating change as depicted in the right change impact matrix in Table 6.5.

Decisions made interactively building the decision tree are alternating decisions of the following two types:

1. Decide for which candidate change the impact is determined

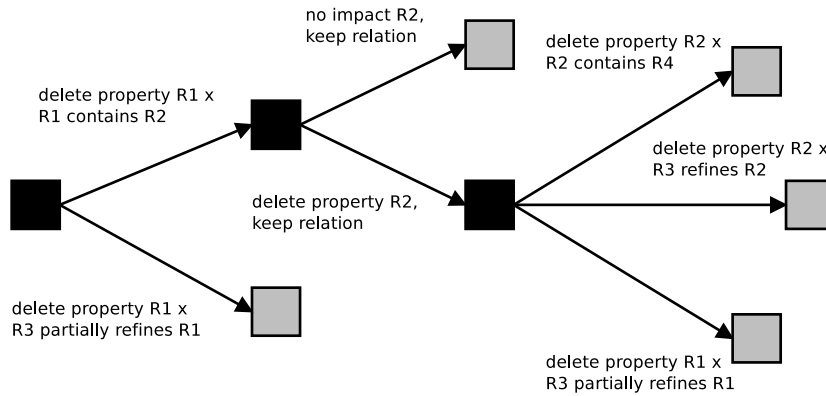


Figure 6.6: An example partially built interactive decision tree for the initial change  $R_1 \xrightarrow{pt} R_1^l$

## 2. Decide the propagated change for the selected candidate change

Once the requirements engineer is done with analyzing multiple possibilities of change impact propagation, he can select a specific visited decision node or end node, and use the sequence of made decisions as the determined proposed/propagated changes.

### 6.4.3 Propagation over given relations only

TRIC provides inferring new relations from given relations by using the semantics of relations. In the design of change propagation, inferred relations are not considered for the following reasons:

1. There is a dependency relation between given and inferred requirements relations. When a given relation is removed, the depending inferred relation is no longer valid. This indicates that given relationships and inferred relationships can not be treated equally during change impact analysis.
2. Consistency checking and inferring relationships are considered activities performed during the modeling of the requirements. During requirements modeling, the requirements engineer uses the inference engine to identify implicitly defined relationships. If the requirements engineer validates certain inferred relations, the engineer must change these inferred relationships to given ones.

Only explicitly given requirements relations are considered links over which impact can be propagated. If the requirements engineer wants to use inferred relations for change impact analysis, these relations must be modeled explicitly as given relations.

### 6.4.4 Impact alternatives and consistency checking

The original inferencing and consistency checking for TRIC is implemented using the JENA First Order Logic reasoner. The design choice for the determined

impact alternatives in subsection 5.2.4 and section 5.3 is to implement the alternatives as a lookup table in Java. The standard JENA ruleset is not able to properly deal well with the disjunction of ‘alternatives’. Consider for example the following rule which can not be implemented in JENA:

if  $x$  is human, then  $x$  is either male or female

In the case that  $x$  is male, the possibility that  $x$  is a female is deemed false is not captured by monotonic logic, which JENA uses for its rule engine. This means that all disjunctions for change impact alternatives must be rewritten to monotonic logic. Expressing possibilities of nonmonotonic logic in monotonic logic can be done[37], but is considered out of scope for this work.

This design does not modify JENA and implement the change impact alternatives and consistency checking of changes in java.

### 6.4.5 Change impact prediction

Change impact prediction is performed by exhaustively generating all change impact alternatives for each candidate change. This is done by performing an exploration algorithm that determines all possible paths leading from the initially proposed change. For each unique path, given the initial change type, all possible propagation paths are determined.

The paths can be generated by either a depth-first traversal or breadth-first traversal. Because all unique paths are to be generated, there is no optimal solution. For this approach, the time complexity for both algorithms are equal. For the implementation of the change impact prediction the depth-first traversal algorithm is used.

The result of this process is all possible propagations from the initially impacted requirement to all reachable requirements. By consequently analyzing these propagations, it is determined if and how requirements can be impacted.

## 6.5 Implementation

In this section the implementation of the following features is explained:

- Determining and representing change impact prediction results
- Application of analysis and changing the requirements model
- Synchronizing the change impact model and RM through using hash for loading and saving CIA’s

### 6.5.1 Change impact prediction results

CIP is performed by interleaving a depth-first traversal over the requirements relations with the different propagation alternatives. The result of the CIP is all unique paths of change propagations with their exhaustively determined impact possibilities.

Presented in Listing 6.1 is the pseudocode for generating all possible paths through decision trees given a change. The implementation combines the reachability analysis using depth-first traversal with the generation of propagation alternatives. Its input is the following arguments:



**Change:** Cannotated edge representing a change

**Graph:** Graph representation of the requirements model

**Path:** Current stack of determined changes (followed edges)

Its methods are:

**getRelatedEdges( Change, Graph ):** Given a Change and Graph, return all related Changes (edges) of the target Requirement when the impact on target requirement is other than ‘no impact’. This method returns an empty set if the impact on target requirement is ‘no impact’.

**removeKnownEdges( nextChanges, Path ):** Given a list of Changes and the already known Path of visited changes, this method returns the list of changes not resulting in cyclic paths, or capturing already visited edges.

**getImpactAlternatives( Change ):** Given an annotated edge that captures the requirements relation, the direction of the change propagation and the type of change, this method returns the set of change propagation alternatives.

---

```

01: dfsPaths( Change, Graph, Path )
02:   path.add( Change )
03:   nextChanges := getRelatedEdges( Change, Graph )
04:   nextChanges := removeKnownEdges( nextChanges, Path )
05:   foreach( nChange ∈ nextChanges )
06:     ImpactAlternatives = getImpactAlternatives( Change )
07:     foreach( Impact ∈ ImpactAlternatives )
08:       nextChange := pChange.copy
09:       nextChange.setImpact( Impact )
10:       newPath := Path.copy
11:       dfsPaths( nextChange, Graph, newPath )
12:   if( nextChanges = ∅ )
13:     allPaths.add( cPath )

```

---

Listing 6.1: Algorithm for interleaving depth-first search with all possible impact alternatives.

---

After performing CIP all possible impact propagations from the given initial change are captured in ‘allPaths’. The captured possible change propagations are analyzed to determine if requirements are impacted in all paths that contain them to determine the degree of impact for the requirement:

**Ensured:** In each propagation path that contains the requirement, the requirement is impacted

**No impact:** In each propagation path that contains the requirement, the requirement is unimpacted

**Possible impact:** The requirement is both found impacted and unimpacted

The number of generated paths depends on the degree of connectivity of the graph and the different impact propagations. To deal with the great number of unique propagation paths that resulting from the CIP, the results are presented in three steps, becoming increasingly more detailed.

1. Initial overview of all requirements, their degree of impact possibility and their change type predictions.
2. Overview of all propagation paths reaching *one* selected requirement, grouped by change type prediction and requirements contained in the propagation path.
3. Visual representation in which one specific order of propagation paths is displayed.

### 6.5.2 Applying the CIA

After the CIA has been performed, the proposed changes are applied to the requirements model if there is no ensured inconsistency in proposed changes. The requirements engineer should decide if detected possible inconsistencies of changes are actual inconsistencies or not.

The following steps are taken in order to apply the results of the CIA:

1. Update requirements descriptions
2. Validate relations
3. Delete requirements and relations

Requirement descriptions are updated in the same order that the changes are determined in CIA. Effectively, for every change the description of the requirement is updated. The requirements engineer can alter the description and add notes about why that impact is determined. Each requirement and relation is either marked for deletion or to be kept.

When all changes have been applied to the requirement descriptions, relations that have contradicting impacts as a result of the propagation of multiple proposed changes are checked. The requirements engineer reviews the updated related requirements and determines if the relation is still valid. Invalid relations are deleted.

After all relations that have contradicting impacts have been validated, relations that are deemed invalid or otherwise marked for deletion are deleted. Subsequently all requirements marked for deletion are deleted. When the requirements model is updated, the CIA is reverted to one without proposed changes.

### 6.5.3 Matching requirements model with change impact model

A change impact model should be saved to and loaded from XML files. These models are stored separately from the requirements models. To ensure that the CIAs is performed on the correct requirements model after loading the model from storage, a SHA1[38] hash is computed from the requirements and saved

with the change impact model. The hash function takes attributes from all requirements and relations as input and provides a string, the hash, as output.

Upon opening a change impact model, the hash from the requirements model is computed and compared to the hash loaded from the change impact model XML file. Only if the computed and loaded hash match, the change impact model is processed.

## 6.6 Tool usage

The most important features of the tool are described in the following subsections:

- Propose change
- Proposed change propagation
- Display proposed change inconsistencies
- Implement proposed changes in the requirements model
- Change impact prediction of a proposed change

### 6.6.1 Propose change

Changes can be proposed to requirements in the model. This is done through the context menu of the requirements view (right-click on requirement) and selecting ‘Propose change’ from the menu. Figure 6.7 depicts the GUI for change proposal for the initial change of ‘change constraint’ on  $R_{97}$ , which supports the activity of ‘Proposing Changes’ as depicted in the activity diagram in Figure 6.1 in Chapter 6.

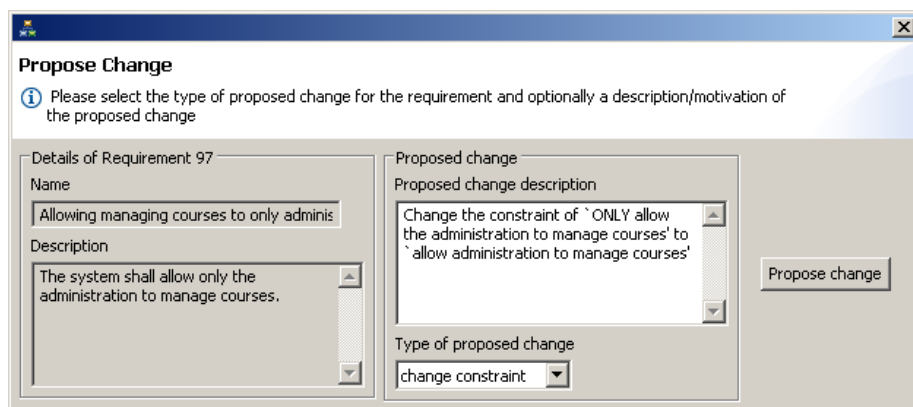


Figure 6.7: Screenshot of ‘Propose Change’ dialog window

The requirements engineer needs to select the appropriate change type in the drop-down list, from possible change type alternatives. Additionally the reason for this proposed change can be noted in the ‘Change Description’ text area.

After proposing the change, the requirements in the requirements view are updated and annotated with the classification of being in the ‘starting impact’ set or ‘candidate impact’ set. The requirements relations fanning out of the changed requirement  $R_{97}$  are marked in the Impact Matrix View as depicted in Figure 6.8. Views similar to the Impact Matrix View are also available in commercial requirements management tools, such as RequisitePro in order to determine the impacted requirements.

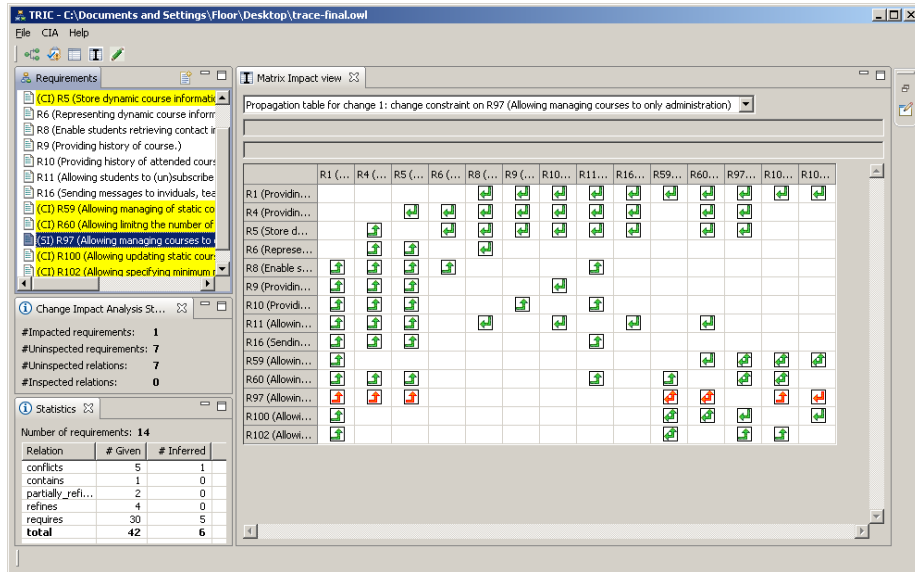


Figure 6.8: Screenshot of tool interface using Matrix Impact View

In the Impact Matrix View, arrows indicate the given relations with their directions. A red arrow indicates an uninspected relation. Because multiple changes can be proposed to the requirements model, a separate Impact Matrix View is kept for each individual proposed change.

The left-hand side of the window lists the requirements of the model with the proposed change requirement ( $R_{97}$ ) tagged as Starting Impact (SI) and (part of the) related requirements ( $R_5$ ,  $R_{59}$ ,  $R_{60}$ ,  $R_{100}$  and  $R_{102}$ ) tagged as Candidate Impact (CI). At the bottom of the left-hand side of the windows, statistics of the CIA such as number of impacted requirements and uninspected requirements are listed.

## 6.6.2 Proposed change propagation

The requirements engineer can select the candidate impacted requirements to propagate the proposed change to these requirements. This is done through either the context menu of the requirements view (right-click on requirement) and selecting ‘Determine proposed impact on requirement’, or double-clicking an uninspected requirements relation in the Impact Matrix View. Figure 6.9 depicts the ‘Determine Proposed Impact’ dialog window for the relation  $R_{97}$  conflicts  $R_{59}$ .

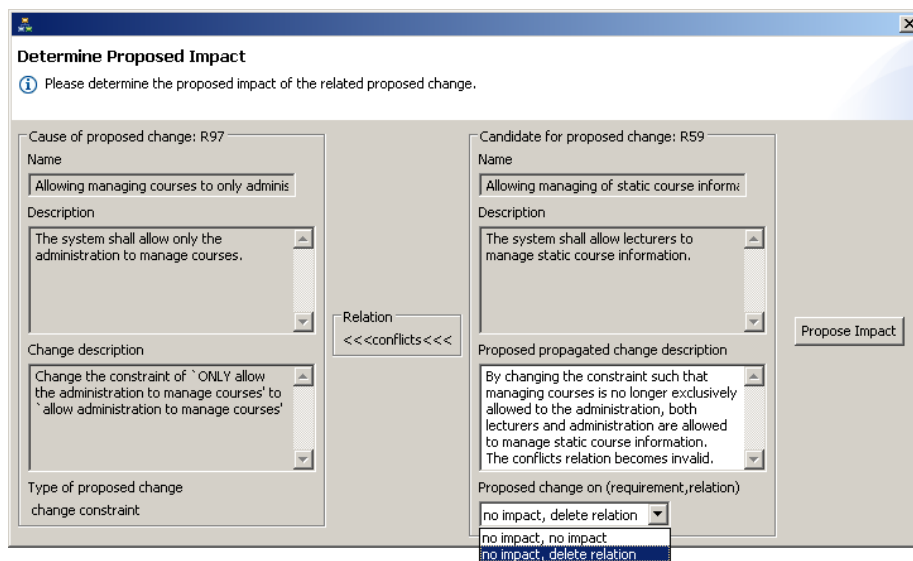


Figure 6.9: Screenshot of ‘Determine Proposed Impact’ window

The requirements engineer has to select the appropriate proposed change for the related requirement, which is selected from the possible impact alternatives given the drop-down list. Motivation for the choice of impact can be given in the ‘Change Description’ text area.

Both the requirements view interface as the Impact Matrix view only allow one specific analysis of proposed changes. Different propagated proposed changes can not be analyzed simultaneously. To support simultaneous analysis of different impact propagations, tool support for building decision trees is provided. Using the Decision Tree on a proposed change allows to explore the decision trees. Figure 6.10 depicts the interface for building the decision tree.

Each node in this interface indicates a decision. The arrows leading to a node indicate decision for each step. The decision tree can be expanded by making decisions. Once analysis using the interactive decision tree is concluded, the requirement engineer can select one path of decisions. By pressing the ‘Use Analysis’ button the decisions captured by the path from the tree root to the selected node are used as the working Proposed Change Propagations. These propagations are then reflected in the CIA. This feature however, is to illustrate that different alternatives can be represented simultaneously, but it lacks functionality such as entering Change Descriptions.

### 6.6.3 Display proposed change inconsistencies

As determined from semantics for classification of change, consistency checking for multiple proposed changes on the same requirement can be performed. Figure 6.11 depicts the initial Impact Inconsistency view.

The initial Impact Inconsistencies view lists the requirements that have contradicting proposed changes. For each requirement, an indication is provided. The inconsistency of changes is denoted as either *ensured* or *possible*. The types

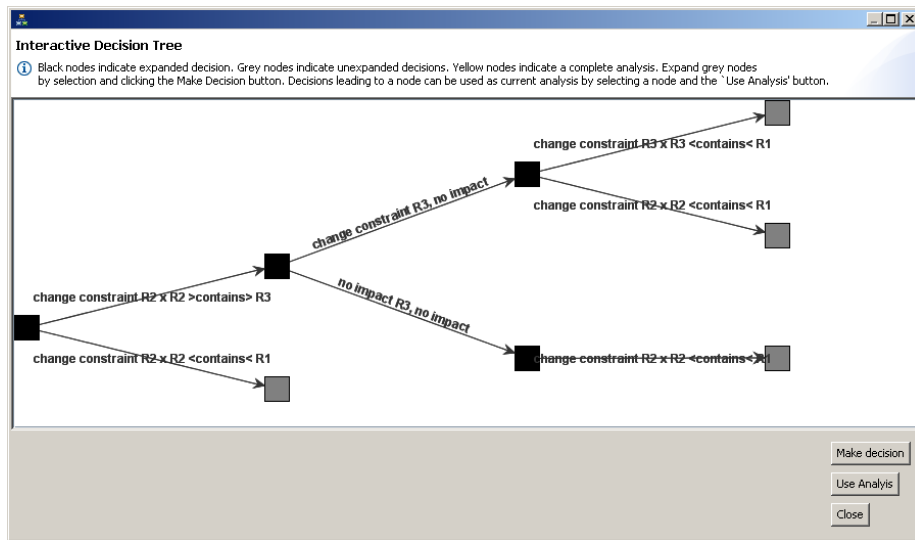


Figure 6.10: Screenshot of 'Interactive Decisiontree' window

| Requirement | Degree   | Contradicting impacts                                     |
|-------------|----------|---|
| R102        | ensured  | impact 1: delete requirement and impact 2: add property   |
| R97         | possible | impact 1: delete property and impact 2: change constraint |
|             |          |   |
|             |          |   |
|             |          |   |
|             |          |   |

Figure 6.11: Screenshot of Impact Inconsistencies window

of impacts leading to the (possible) inconsistency are listed.

The tool provides an explanation for inconsistencies. Consider the example of the contradicting proposed changes 'Change Constraint to Property of Requirement' and 'Delete Property for requirement  $R_{97}$ ' as depicted in Figure 6.12. Each individual propagation path of the proposed change leading to the inconsistency can be visualized.

#### 6.6.4 Implement proposed changes in the model

The tool enables the requirements engineer to implement proposed and propagated proposed changes according to the propagation path. The first proposed change in the path is implemented first. Then, propagated proposed changes are implemented, as depicted in Figure 6.14.

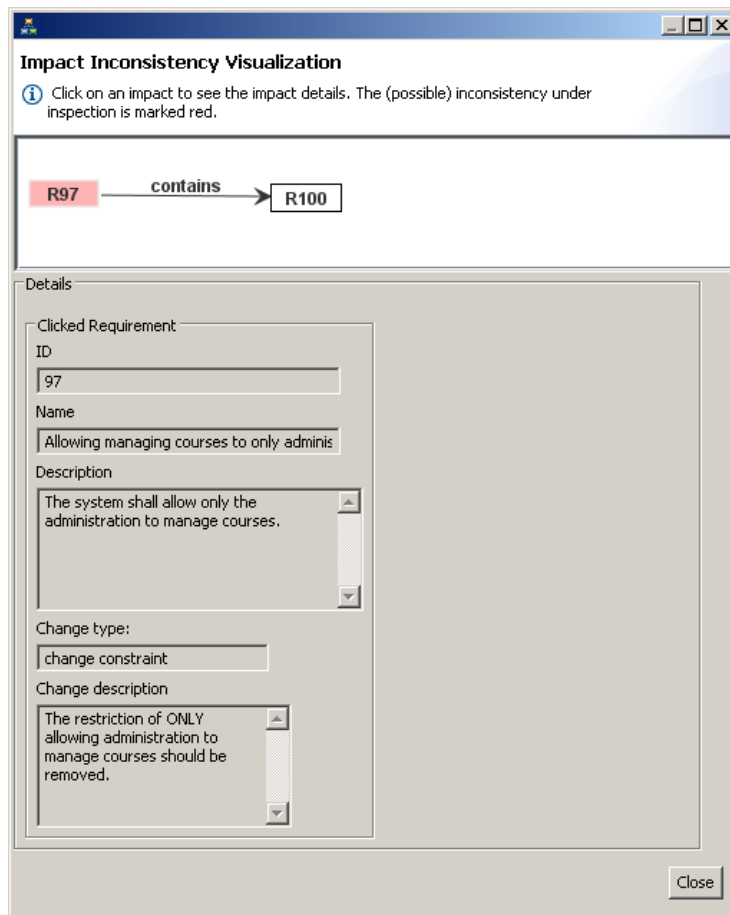


Figure 6.12: Screenshot of Impact Inconsistency Visualization window

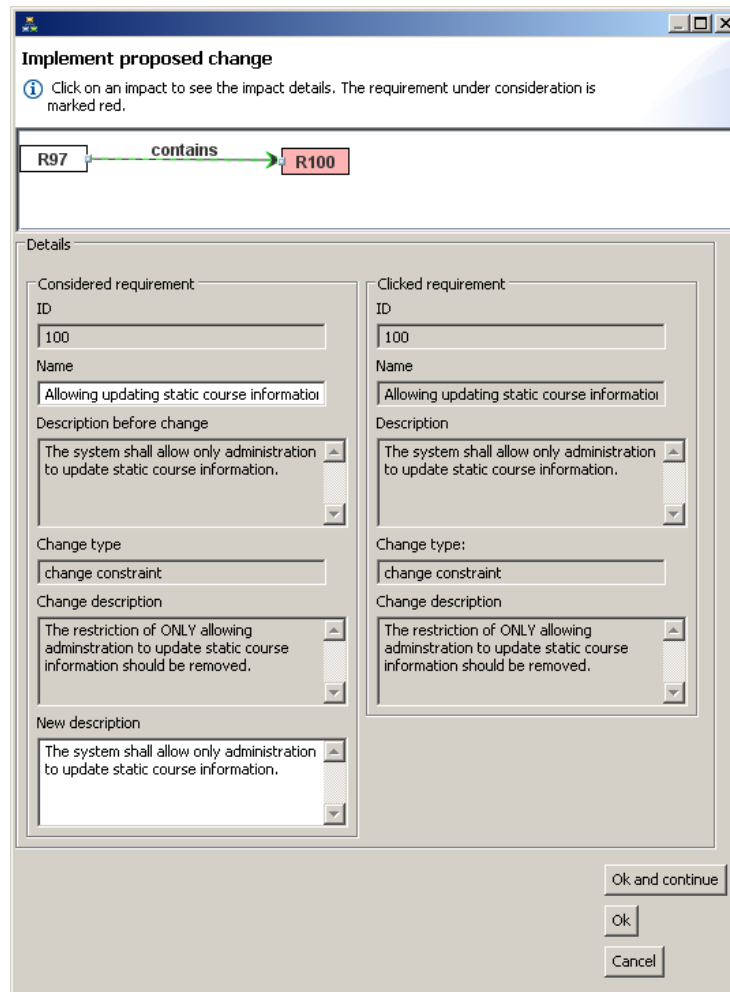


Figure 6.13: Screenshot of Implementing the propagated proposed change



### 6.6.5 Implement proposed changes in the model

The tool enables the requirements engineer to implement proposed and propagated proposed changes according to the propagation path. The first proposed change in the path is implemented first. Then, propagated proposed changes are implemented, as depicted in Figure 6.14.

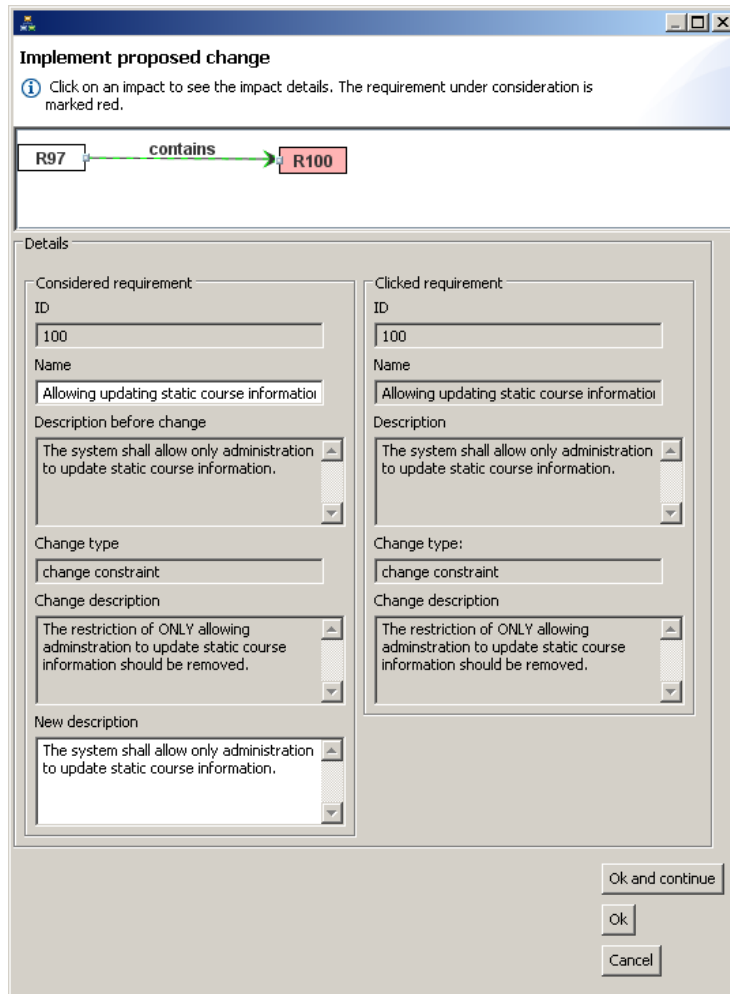
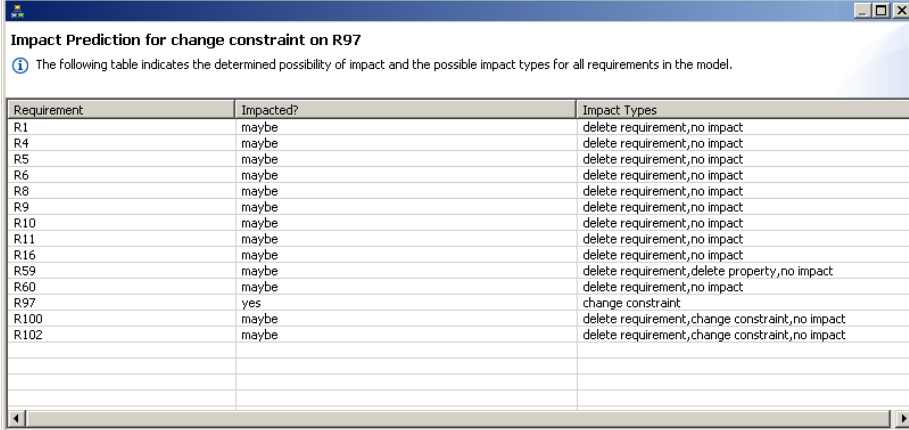


Figure 6.14: Screenshot of Implementing the propagated proposed change

### 6.6.6 CIP of a proposed change

The tool provides impact prediction for a proposed change. All possible propagation paths in the requirements model are traversed in order to determine alternative change types for the propagation. Figure 6.15 depicts an example of the results for performing CIP.



| Requirement | Impacted? | Impact Types                                     |
|-------------|-----------|--|
| R1          | maybe     | delete requirement, no impact                    |
| R4          | maybe     | delete requirement, no impact                    |
| R5          | maybe     | delete requirement, no impact                    |
| R6          | maybe     | delete requirement, no impact                    |
| R8          | maybe     | delete requirement, no impact                    |
| R9          | maybe     | delete requirement, no impact                    |
| R10         | maybe     | delete requirement, no impact                    |
| R11         | maybe     | delete requirement, no impact                    |
| R16         | maybe     | delete requirement, no impact                    |
| R59         | maybe     | delete requirement, delete property, no impact   |
| R60         | maybe     | delete requirement, no impact                    |
| R97         | yes       | change constraint                                |
| R100        | maybe     | delete requirement, change constraint, no impact |
| R102        | maybe     | delete requirement, change constraint, no impact |

Figure 6.15: Screenshot of results of CIP for ‘change constraint’ on  $R_{97}$

The CIP results list all requirements in the model. For each requirement, an indication is given if the requirement is impacted by the proposed change by either *yes*, *no* or *maybe*. The different change impact types are listed in the third column.

The tool also provides the propagation paths per requirement for the impacts listed in the CIP results.

## 6.7 Conclusion

In this chapter tool support has been described. In Section 6.2 the high-level functional requirements are described. Subsection 6.2 describes the intended supported activities.

Section 6.3 describes the high-level architecture of the components. The existing and added components, as well as the altered components are listed and their responsibilities are described.

In Section 6.4 the non-trivial design choices are described. The choices consists of the separation of the change impact model and requirements model, the representation of changes as graph edges and the categorization of visited and unvisited changes. In CIA, only given relationships are considered for propagation, while inferred relations are not used to determine change propagation. The propagation and consistency checking rules are implemented in java, rather than using the reasoner engine JENA.

The implementation as described in section 6.5 mentions the non-trivial implementation choices made.

# Chapter 7

## Evaluation

### 7.1 Introduction

In this chapter CIA using semantics of requirements relations is compared to CIA without using the semantics as provided by common industry standard tools. This is done by using an example case study of performing CIA.

To the best of our knowledge, there is no existing literature on CIA using semantics of traceability relations in requirements models, therefore a comparison of the used approach with literature is omitted.

In section 7.2 the construction of the example case study used for the evaluation is described.

Section 7.3 describes the comparison of performing CIA with semantics of relations with an approach not using this semantics. For comparison a model found in literature on tracing activities is described. Consequently single impact propagation rules are compared between the approaches as well as a comparison of results produced by CIP.

This chapter is summarized and concluded in section 7.4

### 7.2 Example case study

In this section the construction of the example case study is described. First the approach as to how the example model is constructed is described, followed by the construction of the example case study model. Then two example change requests are described, which will be used for the comparison of performing CIA using semantics of change to the ‘traditional’ approach.

#### 7.2.1 Approach

To evaluate CIA with use of semantics of requirements relations, first an example requirements model is constructed. This constructed requirements model is the model used for the case study. The requirements specification document that is used, is the Course Management System (CMS). The textual requirements are formalized in FOL. The composition of the textual requirements is identified using Wasson’s primitives. These primitives are then used to determine the de-

composition of the formalized requirements using properties and their captured constraints.

The actual approach of mapping textual requirements to FOL is considered out of scope for this thesis. It is performed to determine and justify used requirements relations. Pairwise comparison of formalized requirements is performed to determine the relations between requirements.

TRIC is used to model the requirements. Inferencing of requirements relations using TRIC is performed to check if there are any emerging relations that were undetected during identifying requirements relations. Subsequently consistency checking is performed to check the validity of the model.

### 7.2.2 Construction of example model

Explicitly formalizing textual requirements in FOL and performing pairwise comparison of requirements to identify requirements relations are time consuming. Due to limited available time, a subset of the textual requirements of the CMS requirements specification document is used for this case study.

The 14 requirements used from the CMS requirements specification document are listed in Appendix A. Assumptions about the requirements are explicitly listed in Appendix A as well. The decomposition of the used requirements using Wasson's primitives and the identification of the properties and their captured constraints are listed in Appendix B.

Pairwise comparison is performed on the formalized requirements. The resulting relations and their requirements are modeled in TRIC. Consequently the inferencing engine is run. The results are listed in Table 7.1

| Relation          | # Given   | # Inferred |
|-------------------|-----------|------------|
| conflicts         | 5         | 0          |
| contains          | 1         | 0          |
| partially_refines | 2         | 0          |
| refines           | 4         | 0          |
| requires          | 30        | 5          |
| <b>total</b>      | <b>42</b> | <b>5</b>   |

Table 7.1: Statistics reported by TRIC after running the inferencing engine

The results indicate 5 found inferred *requires* relations. Investigation of these inferred relations yielded that 1 *requires* relation is inferred from a *contains* relation. The other 4 *requires* relations are inferred from *refines* relations. Both the *contains* and *refines* relations imply (the weaker) *requires* relation. For performing CIA with semantics, the most stringent relation should be used. These were already explicitly captured by the model, thus there is no need to also capture the inferred *requires* relations.

Running the consistency checker yielded no inconsistencies. The requirements relations resulting from the pairwise comparison are listed in the traceability matrix depicted in Table 7.2. Cells containing a relation indicate the

requirement relation *from* the requirement in the row *to* the requirement in the column, e.g.  $R_5$  *refines*  $R_4$ .

|           | 1  | 4  | 5  | 6  | 8 | 9  | 10 | 11 | 16 | 59  | 60 | 97  | 100 | 102 |
|-----------|----|----|----|----|---|----|----|----|----|-----|----|-----|-----|-----|
| $R_1$     |    |    |    |    |   |    |    |    |    |     |    |     |     |     |
| $R_4$     |    |    |    |    |   |    |    |    |    |     |    |     |     |     |
| $R_5$     |    | rf |    |    |   |    |    |    |    |     |    |     |     |     |
| $R_6$     |    | rf | rq |    |   |    |    |    |    |     |    |     |     |     |
| $R_8$     | rq | rq | rq | rq |   |    |    | rq |    |     |    |     |     |     |
| $R_9$     | rq | rq | rq |    |   |    |    |    |    |     |    |     |     |     |
| $R_{10}$  | rq | rq | rq |    |   | rf |    | rq |    |     |    |     |     |     |
| $R_{11}$  | rq | rq | rq |    |   |    |    |    |    |     |    |     |     |     |
| $R_{16}$  | rq | rq | rq |    |   |    |    | rq |    |     |    |     |     |     |
| $R_{59}$  | rq |    |    |    |   |    |    |    |    |     |    | cf  | cf  | cf  |
| $R_{60}$  | rq | rq | rq |    |   |    |    | rq |    | prf |    | cf  | cf  | cf  |
| $R_{97}$  | rq | rq | rq |    |   |    |    |    |    | cf  | cf |     | ct  |     |
| $R_{100}$ | rq |    |    |    |   |    |    |    |    | cf  | cf |     |     |     |
| $R_{102}$ | rq |    |    |    |   |    |    |    |    | cf  | cf | prf | rf  |     |

Table 7.2: Requirement relations of the constructed requirements model. Contains (ct), conflicts (cf), refines (rf), requires (rq) and partially refines (pq)

### 7.2.3 Change requests

CIA is performed on two change requests. These change requests are such that they represent domain changes. The change requests describe the following changes to the requirements model:

1. The exclusive restriction that *only* administration should be allowed to manage courses should be removed.
2. Lecturers should also be allowed to manage course information that changes while the course is being given.

For both changes, the following actions are performed;

**Identify initially impacted requirement:** The initially impacted requirement is identified in the requirements model, to which the initial change will be proposed.

**Determine classification of change:** The classification of change to the initially changed requirement is determined.

**Derive resulting impacts:** The expected model changes for each change scenario are listed, which are a result of applying the CIA and performing proper change propagation.

#### Change request 1

The property capturing that *only* administration should be allowed to manage courses is captured by requirement  $R_{97}$ . Therefore,  $R_{97}$  is identified as the initially impacted requirement.

The indication that *only* the administration is allowed to do so, is identified as a limitation using Wasson's primitives and as a constraint in the formalized requirement. Considering this constraint should be changed such that the *only* is

removed from the constraint, the classification of change is identified as ‘change constraint’.

The requested domain change is thus mapped to changing the constraints which impose the exclusiveness of the administration as follows (predicates contained in the parentheses indicate constraint that is part of the property) such

that  $R_{97} \xrightarrow{ct \rightarrow ct'} R_{97}^l$ :

$$\begin{aligned} R_{97} &= \text{enable}(x, y) \wedge \text{manage}(x) \wedge \text{course}(y) \wedge ( \text{allow}(x, z) \wedge \\ &\text{administrator}(z) \wedge \neg \text{lecturer}(z) \wedge \neg \text{student}(z) ) \\ R_{97}^l &= \text{enable}(x, y) \wedge \text{manage}(x) \wedge \text{course}(y) \wedge ( \text{allow}(x, z) \wedge \\ &\text{administrator}(z) ) \end{aligned}$$

The constraint explicitly captures that *only* administration is allowed, such that the operation is not allowed to the other users of the system; lecturer and student. To apply the domain change, the constraint should be changed in such a way that both the lecturer and student users are not explicitly disallowed to perform the managing operation. The expected resulting impacts from performing this change are determined by checking where this constraint of *only* allowing administration to create, read, update and/or delete courses occurs. The identified impacted requirements are  $R_{100}$  and  $R_{102}$ . The identified impact type for both requirements is ‘change constraint’:

- Change constraint in  $R_{100}$ , such that its constraint does no longer capture  $\neg \text{student}(z) \wedge \neg \text{lecturer}(z)$
- Change constraint in  $R_{102}$  such that its constraint does no longer capture  $\neg \text{student}(z) \wedge \neg \text{lecturer}(z)$

As a result the conflict between ‘only’ allowing administration to manage courses and (also) allowing lecturers to do so as well is resolved. To reflect this, the conflicts relations should be removed:

- Remove relation  $R_{97} \text{ conflicts } R_{59}$
- Remove relation  $R_{97} \text{ conflicts } R_{60}$
- Remove relation  $R_{59} \text{ conflicts } R_{100}$
- Remove relation  $R_{59} \text{ conflicts } R_{102}$
- Remove relation  $R_{60} \text{ conflicts } R_{100}$
- Remove relation  $R_{60} \text{ conflicts } R_{102}$

## Change request 2

The glossary states that information that changes while the course is being given is considered ‘dynamic course information’. Thus, the change request is such that the requirements should be changed such that lecturers are allowed to manage dynamic course information. Requirement  $R_{59}$  indicates that lecturers should be allowed to manage static course information. The change request is interpreted as changing this requirement in such a way that it also captured dynamic course information to be allowed to be managed by lecturers. Therefore,  $R_{59}$  is identified as the initially impacted requirement.

The type of change of altering the requirement to allow managing of dynamic course information is considered as ‘add property’. The formalized requirement before and after the change, such that  $R_{59} \xrightarrow{+pt} R_{59}^l$ :

$$\begin{aligned} R_{59} &= \text{enable}(x, y) \wedge \text{manage}(x) \wedge \text{course\_information}(y) \wedge \\ &\text{static}(y) \wedge \text{allow}(x, z) \wedge \text{lecturer}(z) \\ R_{59}^l &= \text{enable}(x_1, y_1) \wedge \text{manage}(x_1) \wedge \text{course\_information}(y_1) \wedge \\ &\text{static}(y_1) \wedge \text{allow}(x_1, z) \wedge \text{enable}(x_2, y_2) \wedge \text{manage}(x_2) \wedge \\ &\text{course\_information}(y_2) \wedge \text{static}(y_2) \wedge \text{allow}(x_2, z) \wedge \text{lecturer}(z) \end{aligned}$$

The added property explicitly states that lecturers should be allowed to manage dynamic course information. There are no expected resulting impacted requirements by adding this property.

### 7.3 Comparison of approaches

In this section we compare the use of CIA using semantics of requirements relations to one that does not use semantics of relations. Industry standard tools used to support CIA such as IBM Rational Requisite Pro (RRP)[39][40], Borland CaliberRM[41], Tipteam Analyst[42] and Telelogic DOORS (now Rational DOORS)[43] do not employ semantics of (specific) requirements relations when used for performing CIA[44]. In this comparison, using semantics for requirements relations is compared to the use of a general traceability relation without additional semantics, such as used by RRP.

#### 7.3.1 Approach

To evaluate how CIA with semantics of requirements relations compares to CIA without this semantics the context of as how the model is constructed should be considered as well. First literature on the tracing activity diagram is described. The activities described by this model are used for comparison of the various activities leading up to and including performing CIA.

First the construction of the model is compared. Then the propagation rules of both approaches are compared. Subsequently the results of CIP using the change scenarios are compared for both approached. The comparison is then concluded with emerging features resulting from using semantics of requirements relations that are not found in CIA performed without semantics.

#### 7.3.2 Tracing Activity Model

The tracing activity model as proposed by Heindl and Biff[23][45] is used as the to identify the various activities that entail capturing traceability relations and performing subsequent CIA. The model describes the following activities that are performed to establish and use traceability:

**Trace Specification:** Definition of types of traces that are to be captured and maintained. For example, traces between requirements and design artifacts or/and traces between requirements themselves.

**Trace Generation:** The activity of identifying and explicitly capturing traces between artifacts to the trace specification. One way to capture the traces is by using traceability matrices.

**Trace Usage:** Using the traces as input of tracing applications like change impact analysis, change impact prediction or consistency checking.

**Trace Validation:** Checking if the existing traceability information is valid or needs to be updated.

**Trace Rework:** The actual adjusting of the artifacts and relations to reflect these updates.

The activities described by the Trace Activity Model are used in the following subsections as a guideline of what to compare of both approaches.

### 7.3.3 Trace specification and generation

RRP only has one traceability type, which is the general traceability relation. The relation is such that it indicates direction, to detect cyclic paths. The relation can be denoted as a tuple of (trace from, trace to). Thus the relation is such that if  $R_1$  traces to  $R_2$  then  $R_2$  traces from  $R_1$ .

The requirements relations *contains*, *refines*, *partially refines*, *requires* and *conflicts* are based on literature study[2]. It is assumed that each traceability relation is interpreted as one of these requirements relations.

As a result, the relations detected by the pairwise comparison can be interpreted as generic trace relations used by RRP, with an exception of the *conflicts* relation, which would cause a cyclic trace. The resulting traceability matrix for use of RRP for the example model is depicted in Table 7.3.

|           | 1 | 4 | 5 | 6 | 8 | 9 | 10 | 11 | 16 | 59 | 60 | 97 | 100 | 102 |
|-----------|---|---|---|---|---|---|----|----|----|----|----|----|-----|-----|
| $R_1$     |   |   |   |   |   |   |    |    |    |    |    |    |     |     |
| $R_4$     |   |   |   |   |   |   |    |    |    |    |    |    |     |     |
| $R_5$     |   | × |   |   |   |   |    |    |    |    |    |    |     |     |
| $R_6$     |   | × | × |   |   |   |    |    |    |    |    |    |     |     |
| $R_8$     | × | × | × | × |   |   |    | ×  |    |    |    |    |     |     |
| $R_9$     | × | × | × | × |   |   |    |    |    |    |    |    |     |     |
| $R_{10}$  | × | × | × | × |   | × |    | ×  |    |    |    |    |     |     |
| $R_{11}$  | × | × | × | × |   |   |    |    |    |    |    |    |     |     |
| $R_{16}$  | × | × | × | × |   |   |    | ×  |    |    |    |    |     |     |
| $R_{59}$  | × |   |   |   |   |   |    |    |    |    |    |    |     |     |
| $R_{60}$  | × | × | × | × |   |   |    | ×  |    | ×  |    |    |     |     |
| $R_{97}$  | × | × | × | × |   |   |    |    |    | ×  | ×  |    | ×   |     |
| $R_{100}$ | × |   |   |   |   |   |    |    |    | ×  | ×  |    |     |     |
| $R_{102}$ | × |   |   |   |   |   |    |    |    | ×  | ×  | ×  | ×   |     |

Table 7.3: Traceability relations of the constructed requirements model. × indicates a trace relation *from* the requirement in the row *to* the requirement in column

The trace specification as employed by RRP is straightforward and uncomplicated. The requirements engineer can intuitively understand and apply the notion of the traceability relation, without required knowledge of additional semantics.

When constructing the model using semantics of requirement relations, it is not only determined *if* requirements are related to eachother, but also *how*



they are related to each other. To understand the different ways in which requirements are related to each other specifically, knowledge of the semantics of relations is required. This imposes additional effort to using specific requirements relations.

By investing this additional effort, the requirements model becomes more detailed as to *how* different requirements are related to each other. It allows for inferencing and inconsistency checking of requirements relations in the model[2].

### 7.3.4 Performing CIA

Performing step by step CIA is started by marking initially changed requirements. Current tool support such as RRP does not consider a classification of change. As a result the initial change is indicated as ‘changed’. Consequently traceability information is used to determine which requirements should be checked. Relations connected to the initially changed requirement are marked for inspection.

Introducing the initial change while using semantics of requirements relations requires determining a change type for the initial change. The requirements engineer therefore requires knowledge about the semantics of change types before being able to identify the initial change type. This requires additional effort compared to using RRP.

#### Propagation of change

Without semantics of relations there is no notion of classification of change. As such, a mapping from a requirement change to a change classification is not needed. Without the classification of change, the change impact alternatives for related requirements are limited. The classification of change has two categories, ‘changed’ and ‘not changed’. As a result, the impact alternatives are limited as well. By following a trace relation the related requirement is either changed, or not. To determine change propagation using RRP, the engineer has to manually determine if the related requirement should be changed. If the related requirement should be changed, the engineer manually determines in what way the requirement should be changed..

Additionally, due to lack of semantics, trace relations should either be kept or removed. From these categories, change impact alternative are derived for CIA as performed by using RRP, without semantics for the ‘change’ requirement change and depicted in Table 7.4.

|    | <b>Change</b> | $R_x$ trace to $R_y$   | $R_x$ trace from $R_y$  |
|----|---------------|--|---|
| a. | Change $R_x$  | (NI,NI)   (Change $R_y$ , NI)  <br>(NI, DR)   (Change $R_y$ ,DR) | (NI,NI)   (Change $R_y$ , NI)  <br>(NI, DR)   (Change $R_y$ , DR) |
| b. | Change $R_y$  | (NI,NI)   (Change $R_x$ , NI)  <br>(NI, DR)   (Change $R_x$ ,DR) | (NI,NI)   (Change $R_x$ , NI)  <br>(NI, DR)   (Change $R_x$ , DR) |

Table 7.4: Change impact alternatives for ‘trace to’ and ‘trace from’ relations without additional semantics

From Table 7.4 it is interpreted that the requirements engineer is not guided if the related requirement is impacted. Propagation alternatives are the same

for every case. In the case that a related requirement is changed, there is no support for determining how the requirement should be changed.

The change impact alternatives resulting from using semantics of requirements relations listed in Tables 5.1 and 5.2 in Chapter 5, provide guidance for the requirements engineer. The change impact alternatives determine *if* propagation of change can occur at all. In some cases propagation of change is ensured, consider for example Change i. Case 1:  $delR_2 \times R_1$  contains  $R_2$  which provides the ensured propagation of deleting a property from  $R_1$ ,  $R_1 \xrightarrow{-pt} R_1^l$ . In some other cases absence of propagation of change is ensured, consider for example Change c. Case 3:  $R_1 \xrightarrow{+pt} R_1^l \times R_1$  partially refines  $R_2$ , which ensured absence of propagation. When multiple impact alternatives are derived, the engineer needs to determine which choice is applicable in that specific case. The derived impact alternatives guide the engineer in determining *if* and *how* the related requirement should be changed.

### Change impact prediction

CIP without employing additional semantics is performed by doing a reachability analysis[1]. Similarly, when performing CIP in RRP a reachability analysis is performed. To determine the CIP using RRP a reachability analysis is performed on the requirement model described by Table 7.3. The result using distance indicators is listed in Table 7.5. From this table it is concluded that all requirements are reachable from any other requirement.

|           |   |   |   |   |   |   |    |    |    |    |    |    |     |     |
|-----------|---|---|---|---|---|---|----|----|----|----|----|----|-----|-----|
|           | 1 | 4 | 5 | 6 | 8 | 9 | 10 | 11 | 16 | 59 | 60 | 97 | 100 | 102 |
| $R_1$     |   | 2 | 2 | 2 | 1 | 1 | 1  | 1  | 1  | 1  | 1  | 1  | 1   | 1   |
| $R_4$     | 2 |   | 1 | 1 | 1 | 1 | 1  | 1  | 1  | 2  | 1  | 1  | 2   | 2   |
| $R_5$     | 2 | 1 |   | 1 | 1 | 1 | 1  | 1  | 1  | 2  | 1  | 1  | 2   | 2   |
| $R_6$     | 2 | 1 | 1 |   | 1 | 2 | 2  | 2  | 2  | 3  | 2  | 2  | 3   | 3   |
| $R_8$     | 1 | 1 | 1 | 1 |   | 2 | 2  | 1  | 2  | 2  | 2  | 2  | 2   | 2   |
| $R_9$     | 1 | 1 | 1 | 2 | 2 |   | 1  | 2  | 2  | 2  | 2  | 2  | 2   | 2   |
| $R_{10}$  | 1 | 1 | 1 | 2 | 2 | 1 |    | 1  | 2  | 2  | 2  | 2  | 2   | 2   |
| $R_{11}$  | 1 | 1 | 1 | 2 | 1 | 2 | 2  |    | 1  | 2  | 1  | 2  | 2   | 2   |
| $R_{16}$  | 1 | 1 | 1 | 2 | 2 | 2 | 2  | 1  |    | 2  | 2  | 2  | 2   | 2   |
| $R_{59}$  | 1 | 2 | 2 | 3 | 2 | 2 | 2  | 2  | 2  |    | 1  | 1  | 1   | 1   |
| $R_{60}$  | 1 | 1 | 1 | 2 | 2 | 2 | 2  | 1  | 2  | 1  |    | 1  | 1   | 1   |
| $R_{97}$  | 1 | 1 | 1 | 2 | 2 | 2 | 2  | 2  | 2  | 1  | 1  |    | 1   | 1   |
| $R_{100}$ | 1 | 2 | 2 | 3 | 2 | 2 | 2  | 2  | 2  | 1  | 1  | 1  |     | 1   |
| $R_{102}$ | 1 | 2 | 2 | 3 | 2 | 2 | 2  | 2  | 2  | 1  | 1  | 1  | 1   |     |

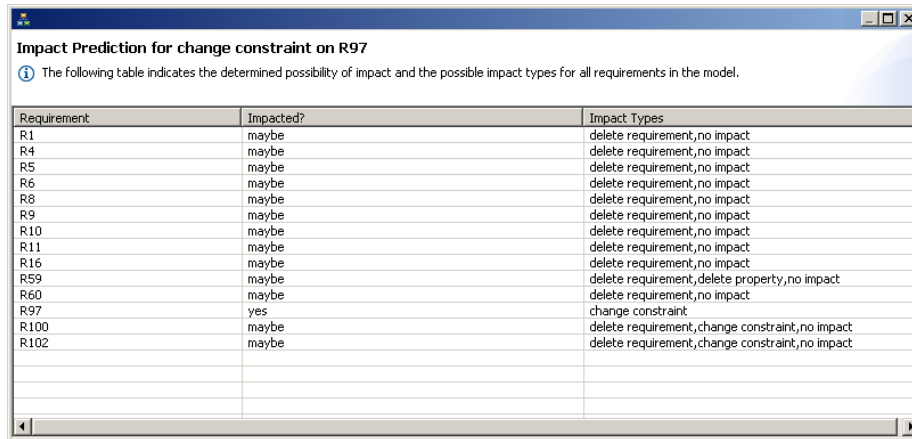
Table 7.5: Reachability matrix with distance indicators

Based on the reachability analysis results depicted in table 7.5 it is determined that CIP performed for both example change requests 1 and 2 yield the same results; all requirements captured by the model are predicted to be impacted.

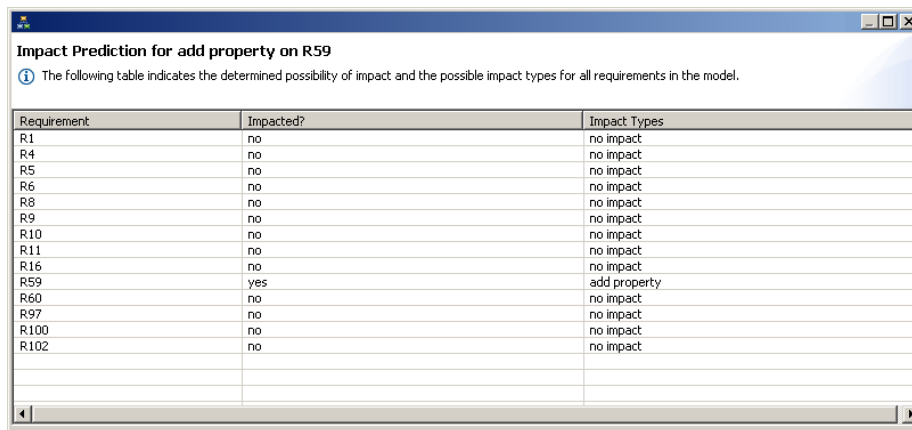
Using CIP based on semantics of requirements relations as described in subsection 5.2.6 for each requirement the degree of impact possibility can be determined, as well as the different types of impacts.

The results of performing CIP using semantics of change type and relations, using tool support are depicted in Figures 7.1 and 7.2.

The following benefits using semantics of requirements relations are identified from comparing the CIP results:



| Requirement | Impacted? | Impact Types                                   |
|-------------|-----------|--|
| R1          | maybe     | delete requirement,no impact                   |
| R4          | maybe     | delete requirement,no impact                   |
| R5          | maybe     | delete requirement,no impact                   |
| R6          | maybe     | delete requirement,no impact                   |
| R8          | maybe     | delete requirement,no impact                   |
| R9          | maybe     | delete requirement,no impact                   |
| R10         | maybe     | delete requirement,no impact                   |
| R11         | maybe     | delete requirement,no impact                   |
| R16         | maybe     | delete requirement,no impact                   |
| R59         | maybe     | delete requirement,delete property,no impact   |
| R60         | maybe     | delete requirement,no impact                   |
| R97         | yes       | change constraint                              |
| R100        | maybe     | delete requirement,change constraint,no impact |
| R102        | maybe     | delete requirement,change constraint,no impact |

Figure 7.1: CIP results for ‘change constraint’ on  $R_{97}$ 


| Requirement | Impacted? | Impact Types |
|-------------|-----------|--------------|
| R1          | no        | no impact    |
| R4          | no        | no impact    |
| R5          | no        | no impact    |
| R6          | no        | no impact    |
| R8          | no        | no impact    |
| R9          | no        | no impact    |
| R10         | no        | no impact    |
| R11         | no        | no impact    |
| R16         | no        | no impact    |
| R59         | yes       | add property |
| R60         | no        | no impact    |
| R97         | no        | no impact    |
| R100        | no        | no impact    |
| R102        | no        | no impact    |

Figure 7.2: CIP results for ‘add property’ on  $R_{59}$ 

- CIP yields a set of impacted requirements that is smaller than all reachable requirements in non-worst-case scenarios, as indicated by the CIP results depicted in Figure 7.2. This shows that CIP using semantics of requirements relations reduces the problem of explosion of impacts.
- CIP provides a change impact type prediction given for each requirement. Therefore CIP using semantics of requirements relations provides more precise results. Only the possible change types are indicated, whereas the possibility of other change types are ensure to be impossible.

The worst-case time complexity of the algorithm for CIP using semantics of requirements relations is  $O(n!)$  for a complete graph, with  $n$  vertices. The worst-case time complexity of a shortest path algorithm of  $O(n \log(n))$ , with  $n$  vertices. It is concluded that the algorithm to perform CIP using semantics of requirements relations scales worse than the CIP algorithm that performs a

reachability analysis.

### 7.3.5 Trace validation

RRP does not support trace validation. Traceability relations do not capture semantical information about the related requirements. Due to the lack of additional semantics about the trace relations, it can not be determined if or not a trace relation becomes invalid due to a change. As a result trace relation validation needs to be performed manually. For each determined impact, the engineer has to manually determine if the trace relation is still valid after the change.

Performing CIA using semantics of requirements relations provides semi-automatic validation of relations. The change impact alternatives are such that they determine the impacts on both the related requirement and the relation. Some cases are unsupported for change validation. Consider the following change impact alternatives case:

#### Example: No support for trace validation

Consider Change g. Case 2:  $R_1 \xrightarrow{-ct} R_1^l \times R_1$  *refines*  $R_2$  in Table 5.1. The following four alternatives are provided:

1. (NI,NI)
2. (NI,DR)
3. ( $R_2 \xrightarrow{-ct} R_2^l$ ,NI)
4. ( $R_2 \xrightarrow{-ct} R_2^l$ ,DR)

Which is the same as the change impact alternatives provided by RPP, apart from the added change impact type. The requirements engineer has to determine manually which choice is applicable in the specific case.

Other cases provide trade validation. Consider the following change impact alternatives case:

#### Example: support for trace validation

Consider Change j. Case 1:  $R_2 \xrightarrow{+pt} R_2^l \times R_1$  *contains*  $R_2$  in Table 5.1. The following two alternatives are provided:

1. (NI,DR)
2. ( $R_1 \xrightarrow{+pt} R_1^l$ ,NI)

The requirements engineer can either determined to propagate the change or not. In the case that the change is not propagated, it is ensured that the relation is no longer valid. In the case that the change is propagated, it is ensured that the relation remains valid.

### 7.3.6 Trace rework

As described in section 6.6, implemented tool support for CIA using semantics of requirements relations allows the proposal of change and subsequent propagation

of proposed changes. The requirements model is not altered until the result of the CIA is applied to the model. By accepting the result and consequently applying the proposed changes, the requirements model is changed.

When multiple different impacts are determined for the affected requirements relations, the tool provides the possibility to the engineer to determine if the relations are still valid or not.

Many industry standard tools in general and RRP in particular do not support change proposal. Using RPP, the support for CIA is triggered as soon as actual changes are made to the requirements model.

## 7.4 Conclusion

This chapter provided an evaluation of performing CIA using semantics of requirements. This is done by comparing the the construction of an example model using semantics of requirements relations and subsequently performing CIA to an approach without this semantics.

Using semantics of requirements relations for CIA is a trade-off. It requires additional effort to construct the requirement model compared to using generic traceability information. Additional time is spent identifying the specific requirements relations. To perform CIA, additional effort is also needed. The classification of change needs to be identified for a requirement change, before CIA using semantics of requirement relations can be performed. However, this investment during requirements modeling leads to benefits in further CIA activities.

Using semantics of requirement relations for CIA provides propagation alternatives. By providing the impact alternatives the requirements engineer is provided with clear alternatives which to chose from, which guides him in decisionmaking. By having clearly defined impact alternatives the requirements engineer is supported. Additionally, using semantics of requirement relations allows for semi-automated requirements validation while propagating changes. As illustrated, in some cases the validation of the requirements relation is implicitly determined by choices of impact alternatives.

When performing CIP the problem of change impact explosion remains the same in the worst case scenario. In other scenarios the result of predicted impacted requirements resulting from performing CIP is a subset of all reachable requirements. In this regard, using semantics of requirements relations as traceability information does counter the change impact explosion. The results yielded by CIP provide more detailed information as to how the requirements are impacted.



# Chapter 8

## Conclusion

### 8.1 Introduction

This chapter concludes the thesis. Section 8.2 summarizes the research performed in the thesis. In section 8.3 the research questions are answered. Section 8.4 describes the future work.

### 8.2 Summary

Eliciting software requirements is one of the first steps in developing software systems. From the moment that the first software artifacts are created, they are subjected to change. To determine what the impact of changes on existing software artifacts are, change impact analysis (CIA) is performed.

Current tool support does not employ semantics for specific relations between requirements. As a result, impact analysis performed suffers from the problem of ‘explosion of impacts’. Bohner states that additional semantics should be employed to counter this problem[1], as described in Chapter 2.

Research performed by Göknil et al. in the Quality-Driven Requirement Engineering and Architectural Design (QuadREAD) project yielded a requirements meta-model with well-defined semantics for requirements and their relations[2]. Chapter 3 describes the formalization of these requirements and relations in First Order Logic (FOL). The relations can be categorized as extensionally (such as the ‘requires’ and ‘conflicts’ relation) and intentionally defined (such as the ‘contains’, ‘refines’ and ‘partially refines’ relation). ‘Tool for Requirements Inferencing and Consistency Checking’ (TRIC) has been developed for automatic reasoning over requirements relations. However, the previous version of TRIC did not support change impact analysis (CIA).

The lack of support for CIA led to the main research question of this thesis: *‘Can CIA using semantics of traceability information in requirements models be provided with tool support’*.

In Chapter 4, literature on the composition of textual requirements is presented. The elements that Heninger[26] and Wasson[3] describe, are mapped to formalization of textual requirements of by Göknil et al. By using the structure and formalization of a textual requirement, a classification of requirement changes with formal semantics is provided. The change classification does not

express itself about why the change is applied. To identify the rationale of change, sources that lead to software evolution are identified using literature. Domain changes and refactoring are considered within the scope of this thesis. These are in turn both formalized in FOL.

Chapter 5 covers the propagation of change. From the working definition of impact as ‘the needed change of software artifacts caused by a change made in software artifacts’, it is interpreted that domain changes drive the change impact analysis. By having clear semantics for the propagation of change, change impact alternatives for domain changes can be derived as described in subsection 5.2.4. Separation of needed change and desired change allows the propagation of change to cover *only* the needed change. Refactoring the model, e.g. the changes not considered needed, is performed separately. Using the semantics of requirements relations and change propagation, relation validation can be supported, as described in subsection 5.2.7. Change impact prediction (CIP) is performed by exhaustively generating all possible change propagations using all different change impact alternatives. The result of the CIP provides additional information about *how* requirements are impacted, as well as an indication *if* requirements are impacted. By using the semantics for classification of change, possible and insured inconsistencies can be automatically identified when multiple impacts are determined for requirements. The main limitation of the approach is the lack of propagation over the conflicts relation and the imprecise impact propagation over the requires relation. These are caused due to not being intentionally defined.

Chapter 6 describes the requirements for performing CIA using the semantics of requirements relations. The requirements are represented in the architecture, and in turn by design and implementation. An important design choice is that the CIA is performed separately from the requirements model. This way the analysis can be performed as a whole without altering the requirements model. When the CIA is accepted, proposed changes are applied to the model. Prototype support for analysing multiple different CIA results is implemented through interactively building the decision trees. Although building decision trees lacks the functionality to enter textual description of changes, it provides an interface that allows for comparing multiple different analyses of the same change.

CIA with semantics of requirements relations is evaluated in Chapter 7. By doing a comparison of CIA with semantics of requirements relations and without semantics, it is determined that it requires additional effort to construct the requirement models and perform subsequent CIA. Investing additional effort provides guidance during change impact propagation and allows for a more precise CIP. The example change scenario indicates that worst-case scenario CIP still yields all related requirements as the impacted set. Even in a worst-case scenario CIP provides additional information with regards to the impact types. In other scenarios the explosion of impacts is reduced. From this it is concluded that the problem of ‘change impact explosion’ is reduced.

### 8.3 Answers to the research questions

The main goal of this research was to provide tool support that uses the semantics of requirements relations to perform CIA. The main research question



therefore was:

*Can CIA using semantics of traceability information in requirements models be provided with tool support?*

As described in Chapter 6 and evaluated in Chapter 7, tool support is developed to perform CIA by using semantics of requirements relations. However, to determine how CIA based on semantics of requirements relations was performed, it first needed to be determined in what way the formalized requirements could be changed. This led to the question:

*What is the change classification for requirements of formalized requirements model?*

As described in Chapter 4, it is identified that the formalized requirements, in addition to properties, also capture constraints. The change classification is determined by these two elements. These elements can be added to, changed in and deleted from requirements. Additionally requirements can be added to or removed from the model. The classification of changes is formalized using FOL as described in section 4.3.

By knowing the classification of changes that can be applied to the requirements, the question is posed as to what the impacts of these changes are. This leads to the question:

*What are the propagation results using this classification of change?*

The classification of changes does not capture why changes are made to the model. In Chapter 5 impact is interpreted as *needed* change, and it is determined that only domain changes cause *needed* change. Rules for change propagation as described in subsection 5.2.2 are determined based on the formalization of domain change and formalization of the requirements relation. A systematical case analysis is performed that covers all cases of propagation of change types over requirements relations. For each case the propagation alternatives are determined. The change impact alternatives are listed in Tables 5.1 and 5.2.

By having determined the change impact alternatives for each impact case, it should be determined how using the semantics of requirements relations for change propagation compared to not using these semantics. This led to the question:

*How does CIA using semantics of traceability information compare to CIA without semantics?*

CIA using semantics of requirements relations is compared to CIA without these semantics in Chapter 7. Knowledge about the semantics of requirements relations is required to construct the requirements model. Constructing a requirements model using specialized requirements relations requires each relation to be classified. Understanding of the classification of change is required to perform CIA using semantics of requirements relations. Using semantics of requirements relations for CIA requires these additional efforts from the requirements engineer, which are not required when performing CIA without semantics of requirements relations.

Investing time in understanding the semantics and constructing a requirements model using these yields the following benefits during CIA:

- Better support for change propagation though provided impact alternatives
- Semi-automated validation of requirements relations
- More precise CIP with indication of impact possibilities and impact change types
- Reduction of the problem of explosion of impacts

## 8.4 Future work

### 8.4.1 Extensionally defined relations

The *requires* and *conflicts* relations are only defined over the sets of systems captured by the requirements, as described in Chapter 3. As illustrated in Chapter 5, by being only extensionally defined, imprecise results may be produced when performing CIA using these semantics. The problem that arises from using the *conflicts* can be countered by making additional modeling decisions. The problem of having *delete requirement* as the only propagation of impact over the *requires* relation however, can not be countered in this way.

Additional research is required to investigate if intentionally defined relations can be identified for these requirements relations, to prevent imprecise results from occurring.

### 8.4.2 Determine scalability

The evaluation performed in this work considers a requirements model of 14 requirements. A requirements model of this size can not be considered comparable to actual requirements models used in practice. CIA using semantics of requirements relations requires more time and additional knowledge to construct a requirements model and propagate the changes. However, it does yield better results.

The time complexity of the CIP algorithm using semantics of requirements relations is  $O(n!)$  for a requirements model that is represented by a complete graph. Additional research should determine how well the CIP algorithm scales for larger requirements models, both in constructing the requirements model and performing the CIA.

### 8.4.3 Extra requirements change impact analysis

Now that CIA can be performed using semantics of requirements models, the set of requirement changes resulting from an initial change can be determined. Following the paradigm of MDE, the next step is to research how these changes can be propagated to other models used in the SE process, using extra-requirements relations. Identifying relations between requirements models and their related components in architectures would allow the results of CIA as yielded by this thesis to be the input for the CIA performed on the architectural models.

Research of semantics of intra-requirement relations should be performed to pave the way for performing CIA using intra-requirement relations.



# Bibliography

- [1] S. Bohner, “Software change impacts - an evolving perspective,” *Software Maintenance, IEEE International Conference on*, vol. 0, p. 263, 2002.
- [2] A. Göknil, I. Kurtev, K. B. van den, and J.-W. Veldhuis, “Semantics of trace relations in requirements models for consistency checking and inferencing,” *Software and Systems Modeling*, vol. Online, December 2009.
- [3] C. S. Wasson, *System analysis, design, and development: Concepts, principles, and practices*. Wiley series in systems engineering and management, Hoboken, NJ: Wiley-Interscience, 2006.
- [4] J. Bézivin and O. Gerbé, “Towards a precise definition of the omg/mda framework,” in *ASE '01: Proceedings of the 16th IEEE international conference on Automated software engineering*, (Washington, DC, USA), p. 273, IEEE Computer Society, 2001.
- [5] A. G. Kleppe, J. Warmer, and W. Bast, *MDA Explained: The Model Driven Architecture: Practice and Promise*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.
- [6] S. Kent, “Model driven engineering,” in *IFM '02: Proceedings of the Third International Conference on Integrated Formal Methods*, (London, UK), pp. 286–298, Springer-Verlag, 2002.
- [7] J.-M. Favre, “Towards a basic theory to model model driven engineering,” *Workshop on Software Model Engineering, WISME 2004, joint event with UML2004*, 2004.
- [8] B. Baudry, C. Nebut, and Y. L. Traon, “Model-driven engineering for requirements analysis,” in *EDOC '07: Proceedings of the 11th IEEE International Enterprise Distributed Object Computing Conference*, (Washington, DC, USA), p. 459, IEEE Computer Society, 2007.
- [9] omg, *Meta Object Facility (MOF) Core Specification Version 2.0*, 2006.
- [10] I. García-Magari no, R. Fuentes-Fernández, and J. J. Gómez-Sanz, “A framework for the definition of metamodels for computer-aided software engineering tools,” *Inf. Softw. Technol.*, vol. 52, no. 4, pp. 422–435, 2010.
- [11] P. Zave, “Classification of research efforts in requirements engineering,” *ACM Comput. Surv.*, vol. 29, no. 4, pp. 315–321, 1997.

- [12] A. van Lamsweerde, "Requirements engineering in the year 00: a research perspective," in *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, (New York, NY, USA), pp. 5–19, ACM, 2000.
- [13] D. T. Ross and K. E. Schoman, "Structured analysis for requirements definition," *IEEE Trans. Softw. Eng.*, vol. 3, no. 1, pp. 6–15, 1977.
- [14] A. Abran, P. Bourque, R. Dupuis, J. W. Moore, and L. L. Tripp, *Guide to the Software Engineering Body of Knowledge - SWEBOK*. Piscataway, NJ, USA: IEEE Press, 2004 version ed., 2004.
- [15] I. Sommerville, *Software Engineering (7th Edition) (International Computer Science Series)*. Addison Wesley, May 2004.
- [16] I. O. Electrical and E. E. (ieee), *IEEE 90: IEEE Standard Glossary of Software Engineering Terminology*. 1990.
- [17] O. C. Z. Gotel and C. W. Finkelstein, "An analysis of the requirements traceability problem," in *Requirements Engineering, 1994., Proceedings of the First International Conference on*, pp. 94–101, 1994.
- [18] R. Wieringa, "An introduction to requirements traceability," tech. rep., University of Vrije, Amsterdam, September 1995.
- [19] C. M. University, "Capability maturity model integration," July 2010. [online] <http://www.sei.cmu.edu/cmml/>.
- [20] F. A. C. Pinheiro, "Requirements traceability," in *Software Requirements, 2003, Kluwer International Series in Engineering and Computer Science, 753*, pp. 91–113, Kluwer, 2003.
- [21] A. G. Dahlstedt and A. Persson, "Requirements interdependencies: state of the art and future challenges," in *Engineering and Managing Software Requirements*, pp. 95–116, Springer-Verlag, 2005.
- [22] B. Ramesh and M. Jarke, "Toward reference models for requirements traceability," *IEEE Transactions on Software Engineering*, vol. 27, pp. 58–93, 2001.
- [23] M. Heindl and S. Biffl, "Modeling of requirements tracing," in *CEE-SET* (B. Meyer, J. R. Nawrocki, and B. Walter, eds.), vol. 5082 of *Lecture Notes in Computer Science*, pp. 267–278, Springer, 2007.
- [24] R. S. Arnold and S. A. Bohner, "Impact analysis - towards a framework for comparison," in *ICSM '93: Proceedings of the Conference on Software Maintenance*, (Washington, DC, USA), pp. 292–301, IEEE Computer Society, 1993.
- [25] A. G. "Tutorial: Requirements relations and definitions with examples."
- [26] K. L. Heninger, "Specifying software requirements for complex systems: New techniques and their application," *IEEE Trans. Softw. Eng.*, vol. 6, no. 1, pp. 2–13, 1980.

- [27] H. Kilov, "From semantic to object-oriented data modeling," in *ICSI* (P. A. Ng, C. V. Ramamoorthy, L. C. Seifert, and R. T. Yeh, eds.), pp. 385–393, IEEE Computer Society, 1990.
- [28] D. E. Perry, "Dimensions of software evolution," in *In Proceedings of the IEEE International Conference on Software Maintenance. IEEE Computer*, pp. 296–303, Society Press, 1994.
- [29] B. P. Lientz and E. B. Swanson, *Software Maintenance Management*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1980.
- [30] A. van Lamsweerde, *Requirements Engineering: From System Goals to UML Models to Software Specifications*. Wiley, March 2009.
- [31] S. Goldsack and A. Finkelstein, "Requirements engineering for real-time systems," *Software Engineering Journal*, vol. 6, no. 3, pp. 101–115, 1991.
- [32] E. Dubois, "Use of deontic logic in the requirements engineering of composite systems," pp. 125–139, 1993.
- [33] J. Veldhuis, "Tool support for a metamodeling approach for reasoning about requirements," April 2009.
- [34] J. McAffer and J.-M. Lemieux, *Eclipse Rich Client Platform: Designing, Coding, and Packaging Java(TM) Applications*. Addison-Wesley Professional, 2005.
- [35] M. Dean and G. Schreiber, "OWL web ontology language reference," W3C recommendation, W3C, February 2004.
- [36] J. Ltd, "Jgraphx (jgraph 6) user manual," September 2010. [online] [http://www.jgraph.com/doc/mxgraph/index\\_javavis.html](http://www.jgraph.com/doc/mxgraph/index_javavis.html).
- [37] J. Meyer and W. van der Hoek, "Non-monotonic reasoning by monotonic means," in *Logics in AI* (J. van Eijck, ed.), vol. 478 of *Lecture Notes in Computer Science*, pp. 399–411, Springer Berlin / Heidelberg, 1991. 10.1007/BFb0018455.
- [38] D. E. Eastlake and P. E. Jones, "Us secure hash algorithm 1 (sha1)." <http://www.ietf.org/rfc/rfc3174.txt?number=3174>.
- [39] IBM, "Rational requisite pro," June 2009. [online] <http://www.rational.com/products/reqpro/>.
- [40] P. Zielczynski, *Requirements Management Using IBM Rational RequisitePro*. IBM Press, 1 ed., 2007.
- [41] Borland, "Requirements management software." <http://www.borland.com/us/products/caliber/index.html>.
- [42] TechnoSolutions, "Topteam analyst for requirements management, requirements management, use cases, use case diagram, use case model, uml use case, uml tool, authoring tool, traceability, actors." [http://www.technosolutions.com/topteam\\_requirements\\_management.html](http://www.technosolutions.com/topteam_requirements_management.html).

- [43] IBM, "Ibm - rational doors - software." <http://www-01.ibm.com/software/awdtools/doors/>.
- [44] M. Abma, "Evaluation of requirements management tools with support for traceability-based change impact analysis," September 2009.
- [45] M. Heindl and S. Biffl, "Requirements tracing strategies for change impact analysis and re-testing," *Technical Report, Technical University of Vienna*, 2007.



## Appendix A

# CMS Requirements Specification Document

This is the subset of the requirements from the Course Management System as referenced in Göknil et al.[2]<sup>1</sup>.

- R1:** The system shall provide *static course information*
- R4:** The system shall provide *dynamic course information*
- R5:** The system shall be able to store *dynamic course information*
- R6:** The system shall be able to represent *dynamic course information*
- R8:** The system shall enable students to retrieve contact information of students and lecturers of subscribed courses
- R9:** The system shall provide the history of a course (view contents of a course over the years)
- R10:** The system shall provide the history of attended courses
- R11:** The system shall enable students to subscribe/unsubscribe to courses
- R16:** The System shall allow sending messages to individuals, teams or all course participants at once
- R59:** The system shall allow lecturers to *manage static course information*
- R60:** The system shall allow lecturers to limit the number of students subscribing to a course
- R97:** The system shall allow only the administration to *manage* courses
- R100:** The system shall allow only the administration to update *static course information*
- R102:** The system shall allow only the administration to specify the minimum number of students for a course. If there are too little subscriptions in a semester, that course will not be given during that semester.

---

<sup>1</sup>[http://wwwhome.cs.utwente.nl/~goknila/sosym/CMS\\_Model2.owl](http://wwwhome.cs.utwente.nl/~goknila/sosym/CMS_Model2.owl)

## Glossary

The following glossary words are present in the original CMS document.

**Static Course Information:** Information of a course which does not change while a course is given, but between semesters. This includes the lecturer, amount of ects and study material.

**Dynamic Course Information:** Information of a course which changes while a course is given. This includes news messages, archived files and roster.

**Manage:** Managing involves the creation, reading, updating and deleting.

### Added to glossary

The following glossary words have been added:

**Attending:** Following a course, and thus being subscribed to that course.

**Course:** A course consists of all information of that course.

**Store:** Retaining something in order to be available for future use.

**Limit:** The greatest possible degree of something.

**Roster:** Refers to the meaning of the Dutch word ‘rooster’ which indicates a schedule of planned lectures, rather than the English definition of roster which places the emphasis on the list of items (and possibly accompanied with the tasks for each name).

## Assumptions

The following assumptions are made about the domain:

1. Students can subscribe to courses before the start of and during semesters.
2. Information can only be represented by the system if it is present in the system, and therefore ‘stored’ in some manner.
3. ‘Stored’ is interpreted as retaining information
4. A roster refers to the Dutch meaning of the word ‘rooster’ which implies a schedule, in this case a schedule of courses during a semester.
5. Students subscribed to a course are considered to be attending (to the lectures of) that course.
6. Students subscribe to courses themselves.
7. ‘Course’ itself, as an object, contains both static and dynamic course information.
8. Course names do not change while the course is given.
9. Limiting the number of subscriptions to a course is interpreted as setting a maximum to the number of subscriptions.

10. Limiting the maximum number of subscriptions to a course is done before the course is given.
11. Setting a minimum number of subscriptions to a course is done before the course is given.



## Appendix B

# Formalization of CMS requirements

**$R_1$  The system shall provide static course information**

**Using Wasson's primitives:**

**Capability:** The system shall provide static course information

**Using FOL formalization:**

**Property:**  $enable(x, y) \wedge provide(x) \wedge course\_information(y) \wedge static(y)$

With the following predicate symbols:

$enable(x, y)$ : stating that functionality  $x$  is provided over  $y$

$provide(x)$ : such that  $x$  is the 'providing' operation

$course\_information(y)$ : such that  $y$  is 'course information'

$static(y)$ : such that  $y$  is 'static'

**$R_4$  The system shall provide dynamic course information**

**Using Wasson's primitives:**

**Capability:** The system shall provide dynamic course information

**Using FOL formalization:**

**Property:**  $enable(x, y) \wedge provide(x) \wedge course\_information(y) \wedge dynamic(y)$

With the following predicate symbols:

$enable(x, y)$ : stating that functionality  $x$  is provided over  $y$

$provide(x)$ : such that  $x$  is the 'providing' operation

$course\_information(y)$ : such that  $y$  is 'course information'

$dynamic(y)$ : such that  $y$  is 'dynamic'

**$R_5$  The system shall be able to store dynamic course information**

**Using Wasson's primitives:**

**Capability:** The system shall be able to store dynamic course information

**Using FOL formalization:**

**Property:**  $enable(x, y) \wedge store(x) \wedge course\_information(y) \wedge dynamic(y)$

With the following predicate symbols:

$enable(x, y)$ : stating that functionality  $x$  is provided over  $y$

$store(x)$ : such that  $x$  is the 'storing' operation

$course\_information(y)$ : such that  $y$  is 'course information'

$dynamic(y)$ : such that  $y$  is 'dynamic'

**$R_6$  The system shall be able to represent dynamic course information**

**Using Wasson's primitives:**

**Capability:** The system shall be able to represent dynamic course information

**Using FOL formalization:**

**Property:**  $enable(x, y) \wedge represent(x) \wedge course\_information(y) \wedge dynamic(y)$

With the following predicate symbols:

$enable(x, y)$ : stating that functionality  $x$  is provided over  $y$

$represent(x)$ : such that  $x$  is the 'represent' operation

$course\_information(y)$ : such that  $y$  is 'course information'

$dynamic(y)$ : such that  $y$  is 'dynamic'

**$R_8$  The system shall enable students to retrieve contact information of students and lecturers of subscribed courses**

**Using Wasson's primitives:**

**Capability:** The system shall enable to retrieve contact information of subscribed courses

**Relational operator 1:** Limited by user type.

**Constraint 1:** To students.

**Relational operator 2:** Limited by participation

**Constraint 2:** If subscribed to the same course

**Using FOL formalization:**

**Property a:**  $enable(x_1, y_1) \wedge allow(x_1, z) \wedge student(z) \wedge course(c) \wedge$   
 $subscribed(z, c) \wedge retrieve(x_1) \wedge contact\_information(y_1) \wedge$   
 $information\_of(y_1, u_1) \wedge lecturer(u_1) \wedge subscribed(u_1, c)$

**Constraint a1:**  $allow(x_1, z) \wedge student(z) \wedge course(c)$

**Constraint a2:**  $subscribed(z, c) \wedge subscribed(u_1, c)$

**Property b:**  $enable(x_2, y_2) \wedge allow(x_2, z) \wedge student(z) \wedge course(c) \wedge$   
 $subscribed(z, c) \wedge retrieve(x_2) \wedge contact\_information(y_2) \wedge$   
 $information\_of(y_2, u_2) \wedge student(u_2) \wedge subscribed(u_2, c)$

**Constraint b1:**  $allow(x_2, z) \wedge student(z) \wedge course(c)$

**Constraint b2:**  $subscribed(z, c) \wedge subscribed(s, c)$

With the following predicate symbols:

$enable(x, y)$ : stating that functionality  $x$  is provided over  $y$

$allow(x, z)$ : stating that functionality  $x$  is allowed to  $z$

$student(z)$ : such that  $z$  is a student

$course(c)$ : such that  $c$  is a course

$subscribed(z, c)$ : stating that object  $z$  is subscribed to  $c$

$retrieve(x)$ : such that  $x$  is the ‘retrieve’ operation

$contact\_information(y)$ : such that  $y$  is contact information

$information\_of(y, u)$ : stating that information of object  $y$  is related to object  
 $u$

$lecturer(u)$ : such that  $u$  is a lecturer

**$R_9$  The system shall provide the history of a course**

**Using Wasson’s primitives:**

**Capability:** The system shall provide the history of a course.

**Using FOL formalization:**

**Property:**  $enable(x, y) \wedge provide(x) \wedge course(y) \wedge history(y)$

With the following predicate symbols:

$enable(x, y)$ : stating that functionality  $x$  is provided over  $y$

$provide(x)$ : such that  $x$  is the ‘providing’ operation

$course(y)$ : such that  $y$  is a course

$history(y)$ : such that  $y$  is history

**$R_{10}$  The system shall provide the history of attended courses**

**Using Wasson's primitives:**

**Capability:** The system shall provide the history of courses

**Relational operator:** Limited by participation

**Constraint:** Those that have attendance

**Using FOL formalization:**

**Property:**  $enable(x, y) \wedge provide(x) \wedge course(y) \wedge history(y) \wedge attended(y)$

**Constraint:**  $history(y) \wedge attended(y)$

With the following predicate symbols:

$enable(x, y)$ : stating that functionality  $x$  is provided over  $y$

$provide(x)$ : such that  $x$  is the 'providing' operation

$course(y)$ : such that  $y$  is a course

$history(y)$ : such that  $y$  is history

$attended(y)$ : such that  $y$  is attended

**$R_{11}$  The system shall enable students to subscribe/unsubscribe to courses**

**Using Wasson's primitives:**

**Capability 1:** The system shall enable to subscribe to courses.

**Capability 2:** The system shall enable to unsubscribe to courses.

**Relational operator:** Limited by user type.

**Constraint:** By students.

**Using FOL formalization:**

**Property a:**  $enable(x_1, y) \wedge subscribe(x_1) \wedge course(y) \wedge allow(x_1, z) \wedge student(z)$

**Constraint a1:**  $allow(x_1, z) \wedge student(z)$

**Property b:**  $enable(x_2, y) \wedge unsubscribe(x_2) \wedge course(y) \wedge allow(x_2, z) \wedge student(z)$

**Constraint b1:**  $allow(x_2, z) \wedge student(z)$

With the following predicate symbols:

$enable(x, y)$ : stating that functionality  $x$  is provided over  $y$

$subscribe(x)$ : such that  $x$  is the 'subscribing' operation



*unsubscribe(x)*: such that  $x$  is the ‘unsubscribing’ operation

*course(y)*: such that  $y$  is a course

*allow(x, y)*: stating that operation  $x$  is allowed to  $y$

*student(z)*: such that  $z$  is a student

**$R_{16}$  The system shall allow sending messages to individuals, teams or all course participants at once**

**Using Wasson’s primitives:**

**Capability:** The system shall allow sending messages to individuals, teams or all course participants at once

**Using FOL formalization:**

**Property a:**  $enable(x) \wedge sending\_message(x, y_1) \wedge recipient(y_1) \wedge individual(y_1)$

**Property b:**  $enable(x) \wedge sending\_message(x, y_2) \wedge recipient(y_2) \wedge team(y_2)$

**Property c:**  $enable(x) \wedge sending\_message(x, y_3) \wedge recipient(y_3) \wedge individual(y_3) \wedge subscribed(y_3, z) \wedge course(z)$

With the following predicate symbols:

*enable(x)*: such that operation  $x$  is provided

*sending\_message(x, y)*: stating that  $x$  is the operation of sending a message to  $y$

*recipient(y)*: such that  $y$  is a recipient of messages

*individual(y)*: such that  $y$  is an individual

*team(y)*: such that  $y$  is a team

*course(z)*: such that  $z$  is a course

*subscribed(y, z)* : stating that  $y$  is subscribed to  $z$

**$R_{59}$  The system shall allow lecturers to manage static course information**

**Using Wasson’s primitives:**

**Capability:** The system shall enable managing static course information

**Relational operator:** Limited by user type.

**Constraint:** By lecturers.

**Using FOL formalization:**

**Property a:**  $enable(x_1, y) \wedge course\_information(y) \wedge static(y) \wedge create(x_1) \wedge allow(x_1, z) \wedge lecturer(z)$

**Constraint a1:**  $allow(x_1, z) \wedge lecturer(z)$

**Property b:**  $enable(x_2, y) \wedge course\_information(y) \wedge static(y) \wedge read(x_2) \wedge allow(x_2, z) \wedge lecturer(z)$

**Constraint b1:**  $allow(x_2, z) \wedge lecturer(z)$

**Property c:**  $enable(x_3, y) \wedge course\_information(y) \wedge static(y) \wedge update(x_3) \wedge allow(x_3, z) \wedge lecturer(z)$

**Constraint c1:**  $allow(x_3, z) \wedge lecturer(z)$

**Property d:**  $enable(x_4, y) \wedge course\_information(y) \wedge static(y) \wedge delete(x_4) \wedge allow(x_4, z) \wedge lecturer(z)$

**Constraint a1:**  $allow(x_4, z) \wedge lecturer(z)$

With the following predicate symbols:

$enable(x, y)$ : stating that functionality  $x$  is provided over  $y$

$course\_information(y)$ : such that  $y$  is course information

$static(y)$ : such that  $y$  is static

$create(x)$ : such that  $x$  is the ‘creating’ operation

$read(x)$ : such that  $x$  is the ‘reading’ operation

$update(x)$ : such that  $x$  is the ‘updating’ operation

$delete(x)$ : such that  $x$  is the ‘deleting’ operation

$allow(x, z)$ : stating that operation  $x$  is allowed to  $z$

$lecturer(z)$ : such that  $z$  is a lecturer

**$R_{60}$  The system shall allow lecturers to limit the number of students subscribing to a course**

**Using Wasson’s primitives:**

**Capability:** The system shall enable limiting the number of students subscribing to a course.

**Relational operator:** Limited by user type.

**Constraint:** By lecturers.

**Using FOL formalization:**

**Property:**  $enable(x, y) \wedge set\_max\_subscriptions(x) \wedge course(y) \wedge allow(x, z) \wedge lecturer(z)$

**Constraint:**  $allow(x, z) \wedge lecturer(z)$

With the following predicate symbols:

$enable(x, y)$ : stating that functionality  $x$  is provided over  $y$

$set\_max\_subscriptions(x)$ : such that  $x$  is the 'setting maximum number of subscriptions' operation

$course(y)$ : such that  $y$  is a course

$allow(x, z)$ : stating that operation  $x$  is allowed to  $z$

$lecturer(z)$ : such that  $z$  is a lecturer

**$R_{97}$  The system shall allow only the administration to manage courses**

**Using Wasson's primitives:**

**Capability:** The system shall enable managing courses

**Relational operator:** Limited by user type.

**Constraint:** Only by administrators.

**Using FOL formalization:**

**Property a:**  $enable(x_1, y) \wedge create(x_1) \wedge course(y) \wedge allow(x_1, z) \wedge administrator(z) \wedge \neg lecturer(z) \wedge \neg student(z)$

**Constraint a1:**  $allow(x_1, z) \wedge administrator(z) \wedge \neg lecturer(z) \wedge \neg student(z)$

**Property b:**  $enable(x_2, y) \wedge read(x_2) \wedge course(y) \wedge allow(x_2, z) \wedge administrator(z) \wedge \neg lecturer(z) \wedge \neg student(z)$

**Constraint b1:**  $allow(x_2, z) \wedge administrator(z) \wedge \neg lecturer(z) \wedge \neg student(z)$

**Property c:**  $enable(x_3, y) \wedge update(x_3) \wedge course(y) \wedge allow(x_3, z) \wedge administrator(z) \wedge \neg lecturer(z) \wedge \neg student(z)$

**Constraint c1:**  $allow(x_3, z) \wedge administrator(z) \wedge \neg lecturer(z) \wedge \neg student(z)$

**Property d:**  $enable(x_4, y) \wedge delete(x_4) \wedge course(y) \wedge allow(x_4, z) \wedge administrator(z) \wedge \neg lecturer(z) \wedge \neg student(z)$

**Constraint d1:**  $allow(x_4, z) \wedge administrator(z) \wedge \neg lecturer(z) \wedge \neg student(z)$

With the following predicate symbols:

$enable(x, y)$ : stating that functionality  $x$  is provided over  $y$

*course(y)*: such that  $y$  is a course

*create(x)*: such that  $x$  is the ‘creating’ operation

*read(x)*: such that  $x$  is the ‘reading’ operation

*update(x)*: such that  $x$  is the ‘updating’ operation

*delete(x)*: such that  $x$  is the ‘deleting’ operation

*allow(x, z)*: stating that operation  $x$  is allowed to  $z$

*administrator(z)*: such that  $z$  is an administrator

*student(z)*: such that  $z$  is a student

*lecturer(z)*: such that  $z$  is a lecturer

**$R_{100}$  The system shall allow only the administration to update static course information**

**Using Wasson’s primitives:**

**Capability:** The system shall enable updating static course information

**Relational operator:** Limited by user type.

**Constraint:** Only by administrators.

**Using FOL formalization:**

**Property:**  $enable(x, y) \wedge update(x) \wedge course\_information(y) \wedge static(y) \wedge allow(x, z) \wedge administrator(z) \wedge \neg lecturer(z) \wedge \neg student(z)$

**Constraint:**  $allow(x, z) \wedge administration(z) \wedge \neg lecturer(z) \wedge \neg student(z)$

With the following predicate symbols:

*enable(x, y)*: stating that functionality  $x$  is provided over  $y$

*course\_information(y)*: such that  $y$  is course information

*static(y)*: such that  $y$  is static

*allow(x, z)*: stating that operation  $x$  is allowed to  $z$

*update(x)*: such that  $x$  is the ‘updating’ operation

*administrator(z)*: such that  $z$  is an administrator

*student(z)*: such that  $z$  is a student

*lecturer(z)*: such that  $z$  is a lecturer

$R_{102}$  The system shall allow only the administration to specify the minimum number of students for a course. If there are too little subscriptions in a semester, that course will not be given during that semester

Using Wasson's primitives:

**Property:**  $enable(x, y) \wedge set\_minimum\_subscriptions(x) \wedge course(y) \wedge allow(x, z) \wedge administrator(z) \wedge \neg lecturer(z) \wedge \neg student(z)$

**Constraint:**  $allow(x, z) \wedge administrator(z) \wedge \neg lecturer(z) \wedge \neg student(z)$

With the following predicate symbols:

$enable(x, y)$ : stating that functionality  $x$  is provided over  $y$

$set\_minimum\_subscriptions(y)$ : such that  $y$  is the operation of 'setting minimum number of descriptions'

$course(y)$ : such that  $y$  is a course

$allow(x, z)$ : stating that operation  $x$  is allowed to  $z$

$administrator(z)$ : such that  $z$  is an administrator

$student(z)$ : such that  $z$  is a student

$lecturer(z)$ : such that  $z$  is a lecturer