

TRACEABILITY
OF
REQUIREMENTS AND SOFTWARE ARCHITECTURE
FOR
CHANGE MANAGEMENT

ARDA GÖKNIL

Ph.D. dissertation committee:

Chairman and secretary:

Prof. Dr. Ir. Anton J. Mouthaan, University of Twente, The Netherlands

Promoter:

Prof. Dr. Ir. Mehmet Akşit, University of Twente, The Netherlands

Assistant promoter:

Dr. Ivan Kurtev, University of Twente, The Netherlands

Members:

Prof. Dr. Roel Wieringa, University of Twente, The Netherlands

Prof. Dr. Ir. Arend Rensink, University of Twente, The Netherlands

Prof. Dr. Richard Paige, University of York, United Kingdom

Prof. Dr. Antonio Vallecillo, University of Malaga, Spain

Prof. Dr. Ir. Paris Avgeriou, University of Groningen, The Netherlands

Dr. Laurent Balmelli, IBM, United States

CTIT Ph.D. thesis series no. 11-192. Centre for Telematics and Information Technology (CTIT),
P.O. Box 217 – 7500 AE Enschede, The Netherlands.

ISSN: 1381-3617

This work has been carried out as part of the QuadREAD project. This project is supported by Nederlandse Organisatie voor Wetenschappelijk Onderzoek (NWO) under the Software Engineering Jacquard program.

ISBN: 978-90-365-3175-7

ISSN: 1381-3617 (CTIT Ph.D. thesis series no. 11-192)

DOI: <http://dx.doi.org/10.3990/1.9789036531757>

Cover design by Kardelen Hatun

Printed by Ipkamp Drukkers B.V., Enschede, The Netherlands

Copyright © Arda Göknal, Enschede, The Netherlands

TRACEABILITY OF REQUIREMENTS AND SOFTWARE ARCHITECTURE FOR CHANGE MANAGEMENT

DISSERTATION

to obtain
the degree of doctor at the University of Twente,
on the authority of the rector magnificus,
Prof. Dr. H. Brinksma,
on account of the decision of the graduation committee,
to be publicly defended
on Friday the 7th of October 2011 at 16.45

by

Arda Göknil

Born on the 24th of December 1980
In Izmir, Turkey

This dissertation is approved by

Prof. Dr. Ir. Mehmet Akşit (promoter)
Dr. Ivan Kurtev (assistant promoter)

Acknowledgments

I would like to thank to my promotor Mehmet Aksit for giving me a chance for Ph.D. I would like to thank to my supervisors Ivan Kurtev and Klaas van den Berg for having the weekly research meetings during my Ph.D.

I would like to thank to the members of my Ph.D. committee: Paris Avgeriou, Laurent Balmelli, Richard Paige, Arend Rensink, Antonio Vallecillo, and Roel Wieringa for spending their time to evaluate my work.

I would like to thank to the members of the software engineering group for the working environment I have had. I am grateful to our secretary Jeanette Rebel-de Boer for her administrative support.

I would like to thank to all friends I have in the Netherlands. They have helped me not to feel homesick. I would like to thank all the people who have contributed to our football matches in TUFAT.

I would like to thank my parents and my sister who stood by me regardless of many obstacles.

Arda Goknil

Enschede, September 2011

Abstract

At the present day, software systems get more and more complex. The requirements of software systems change continuously and new requirements emerge frequently. New and/or modified requirements are integrated with the existing ones, and adaptations to the architecture and source code of the system are made. The process of integration of the new/modified requirements and adaptations to the software system is called *change management*. The size and complexity of software systems make change management costly and time consuming. To reduce the cost of changes, it is important to apply change management as early as possible in the software development cycle. Requirements traceability is considered crucial in change management for establishing and maintaining consistency between software development artifacts. It is the ability to link requirements back to stakeholders' rationales and forward to corresponding design artifacts, code, and test cases. When changes for the requirements of the software system are proposed, the impact of these changes on other requirements, design elements and source code should be traced in order to determine parts of the software system to be changed. Determining the impact of changes on the parts of development artifacts is called *change impact analysis*. Change impact analysis is applicable to many development artifacts like requirements documents, detailed design, source code and test cases. Our focus is change impact analysis in requirements and software architecture.

The need for change impact analysis is observed in both requirements and software architecture. When a change is introduced to a requirement, the requirements engineer needs to find out if any other requirement related to the changed requirement is impacted. After determining the impacted requirements, the software architect needs to identify the impacted architectural elements by tracing the changed requirements to software architecture. It is hard, expensive and error prone to manually trace impacted requirements and architectural elements from the changed requirements. There are tools and approaches that automate change impact analysis like IBM Rational RequisitePro and DOORS. In most of these tools,

traces are just simple relations and their semantics is not considered. Due to the lack of semantics of traces in these tools, all requirements and architectural elements directly or indirectly traced from the changed requirement are candidate impacted. The requirements engineer has to inspect all these candidate impacted requirements and architectural elements to identify changes if there are any. In this thesis we address the following problems which arise in performing change impact analysis for requirements and software architecture.

Explosion of impacts in requirements after a change in requirements. In practice, requirements documents are often textual artifacts with implicit structure. Most of the relations among requirements are not given explicitly. There is a lack of precise definition of relations among requirements in most tools and approaches. Due to the lack of semantics of requirements relations, change impact analysis may produce high number of false positive and false negative impacted requirements. A requirements engineer may have to analyze all requirements in the requirements document for a single change. This may result in neglecting the actual impact of a change.

Manual, expensive and error prone trace establishment. Considerable research has been devoted to relating requirements and design artifacts with source code. Less attention has been paid to relating Requirements (R) with Architecture (A) by using well-defined semantics of traces. Designing architecture based on requirements is a problem solving process that relies on human experience and creativity, and is mainly manual. The software architect may need to manually assign traces between R&A. Manual trace assignment is time-consuming, expensive and error prone. The assigned traces might be incomplete and invalid.

Explosion of impacts in software architecture after a change in requirements. Due to the lack of semantics of traces between R&A, change impact analysis may produce high number of false positive and false negative impacted architectural elements. A software architect may have to analyze all architectural elements in the architecture for a single requirements change.

In this thesis we propose an approach that reduces the explosion of impacts in R&A. The approach employs semantic information of traces and is supported by tools. We consider that every relation between software development artifacts or between elements in these artifacts can play the role of a trace for a certain traceability purpose like change impact analysis. We choose *Model Driven Engineering (MDE)* as a solution platform for our approach. MDE provides a uniform treatment of software artifacts (e.g. requirements documents, software design and test documents) as models. It also enables using different formalisms to reason about development artifacts described as models. To give an explicit structure to requirements documents and treat requirements, architecture and traces in a uniform way,

we use metamodels and models with formally defined semantics. The thesis provides the following contributions:

A modeling language for definition of requirements models with formal semantics. The language is defined according to the MDE principles by defining a metamodel. It is based on a survey about the most commonly found requirements types and relation types. With this language, the requirements engineer can explicitly specify the requirements and the relations among them. The semantics of these entities is given in First Order Logic (FOL) and allows two activities. First, new relations among requirements can be inferred from the initial set of relations. Second, requirements models can be automatically checked for consistency of the relations. Tool for Requirements Inferencing and Consistency Checking (TRIC) is developed to support both activities. The defined semantics is used in a technique for change impact analysis in requirements models.

A change impact analysis technique for requirements using semantics of requirements relations and requirements change types. The technique aims at solving the problem of explosion of impacts in requirements when semantics of requirements relations is missing. The technique uses formal semantics of requirements relations and requirements change types. A classification of requirements changes based on the structure of a textual requirement is given and formalized. The semantics of requirements change types is based on FOL. We support three activities for impact analysis. First, the requirements engineer proposes changes according to the change classification before implementing the actual changes. Second, the requirements engineer identifies the propagation of the changes to related requirements. The change alternatives in the propagation are determined based on the semantics of change types and requirements relations. Third, possible contradicting changes are identified. We extend TRIC with a support for these activities. The tool automatically determines the change propagation paths, checks the consistency of the changes, and suggests alternatives for implementing the change.

A technique that provides trace establishment between R&A by using architecture verification and semantics of traces. It is hard, expensive and error prone to manually establish traces between R&A. We present an approach that provides trace establishment by using architecture verification together with semantics of requirements relations and traces. We use a trace metamodel with commonly used trace types. The semantics of traces is formalized in FOL. Software architectures are expressed in the Architecture Analysis and Design Language (AADL). AADL is provided with a formal semantics expressed in Maude. The Maude tool set allows simulation and verification of architectures. The first way to establish traces is to use architecture verification techniques. A given requirement is reformulated as a property in

terms of the architecture. The architecture is executed and a state space is produced. This execution simulates the behavior of the system on the architectural level. The property derived from the requirement is checked by the Maude model checker. Traces are generated between the requirement and the architectural components used in the verification of the property. The second way to establish traces is to use the requirements relations together with the semantics of traces. Requirements relations are reflected in the connections among the traced architectural elements based on the semantics of traces. Therefore, new traces are inferred from existing traces by using requirements relations. We use semantics of requirements relations and traces to both generate/validate traces and generate/validate requirements relations. There is a tool support for our approach. The tool provides the following: (1) generation/validation of traces by using requirements relations and/or verification of architecture, (2) generation/validation of requirements relations by using traces.

A change impact analysis technique for software architecture using architecture verification and semantics of traces between R&A. The software architect needs to identify the impacted architectural elements after requirements change. We present a change impact analysis technique for software architecture using architecture verification and semantics of traces. The technique is semi-automatic and requires participation of the software architect. Our technique has two parts. The first part is to identify the architectural elements that implement the system properties to which proposed requirements changes are introduced. By having the formal semantics of requirements relations and traces, we identify which parts of software architecture are impacted by a proposed change in requirements. We have extended TRIC for determining candidate impacted architectural elements. The second part of our technique is to propose possible changes for software architecture when the software architecture does not satisfy the new and/or changed requirements. The technique is based on architecture verification. The output of verification is a counter example if the requirements are not satisfied. The counter example is used with a classification of architectural changes in order to propose changes in the software architecture. These changes produce a new version of the architecture that possibly satisfies the new or the changed requirements.

Contents

Table of Figures	xix
Table of Tables	xxiv
Abbreviations	xxv
1 Introduction	1
1.1 Context.....	1
1.2 Problem Statement.....	2
1.3 Research Questions.....	3
1.4 Research Methodology	4
1.5 Approach	5
1.6 Contributions	7
1.7 Outline of the Thesis	9
2 Background and Definitions	13
2.1 Introduction	13
2.2 Requirements Engineering.....	14
2.2.1 Software Requirements.....	14
2.2.2 Requirements Engineering Processes.....	15
2.2.3 Software Requirements Specification and Documentation.....	16

2.3	Software Architecture	17
2.3.1	Definitions of Software Architecture	18
2.3.2	Software Architecture Analysis.....	20
2.3.3	Architectural Patterns and Styles.....	21
2.3.4	Modeling Software Architecture	21
2.4	Software Change Management.....	23
2.4.1	Strategies for Software Change Management	24
2.4.2	Software Maintenance	24
2.4.3	Requirements Evolution.....	26
2.4.4	Change Impact Analysis	26
2.5	Traceability	28
2.5.1	Definitions of Traceability.....	28
2.5.2	Core Concepts of Traceability	29
2.5.3	Classification of Traces	30
2.6	Model Driven Engineering	32
2.6.1	Model Driven Architecture	32
2.6.2	Model Driven Engineering.....	34
2.7	Survey of Traceability in MDE	35
2.7.1	Traceability Approaches in MDE	35
2.7.2	Evaluation of the Approaches	40
2.7.3	Open Issues for Traceability in MDE.....	44
2.8	Conclusions	46
3	Analysis of Impacts Explosion in Traceability	47
3.1	Introduction	47
3.2	Impacts Explosion Problem.....	48
3.3	Impacts Explosion in Requirements and Software Architecture	50
3.4	Change Scenarios for Change Impact Analysis.....	54
3.4.1	Scenario 1: Requirements Evolve	55
3.4.2	Scenario 2: Requirements and Software Architecture Evolve	55
3.5	Summary of the Problems	56

3.6 Conclusions	57
4 Semantics of Requirements Relations	59
4.1 Introduction	59
4.2 Approach	61
4.3 Requirements Metamodel	61
4.4 Formalization of Requirements and Relations	64
4.4.1 Formalization of Requirements	64
4.4.2 Formalization of Requirements Relations	66
4.4.3 Discussion on the Chosen Formalization.....	75
4.5 Inferencing and Consistency Checking	76
4.6 Tool Support.....	88
4.6.1 The Modeling Process.....	88
4.6.2 Tool Architecture.....	89
4.6.3 Tool Features.....	91
4.7 Example: Course Management System	98
4.7.1 Modeling the Requirements	99
4.7.2 Inferring Requirements Relations	101
4.7.3 Checking Consistency	102
4.8 Related Work.....	103
4.8.1 Requirements Relations	103
4.8.2 Requirements Metamodeling	105
4.8.3 Requirements Reasoning	107
4.8.4 Tool Support.....	108
4.9 Conclusions	109
5 Change Impact Analysis in Requirements Models	113
5.1 Introduction	113
5.2 Approach	115
5.3 Classification of Changes in Requirements	116
5.3.1 Structure of a Textual Requirement.....	116

5.3.2	Change Types for Requirements Models.....	119
5.3.3	Rationale of Changes.....	127
5.4	Change Propagation and Change Consistency Checking.....	130
5.5	Discussion on the Approach.....	143
5.6	Tool Support.....	144
5.6.1	The Modeling Process.....	144
5.6.2	Tool Architecture.....	146
5.6.3	Tool Features.....	147
5.7	Example: Course Management System	156
5.7.1	Proposing and Propagating Requirements Changes.....	156
5.7.2	Checking Consistency	159
5.8	Evaluation of the Approach.....	160
5.9	Related Work.....	165
5.9.1	Change Classification with Formal Semantics	165
5.9.2	Change Impact Analysis in Requirements	166
5.9.3	Tool Support.....	168
5.10	Conclusions	169
6	Traces between Requirements and Software Architecture	171
6.1	Introduction	171
6.2	Overview of the Approach.....	174
6.3	Trace Metamodel.....	176
6.4	Formalization of Trace Types	178
6.4.1	Formalization of Requirements.....	179
6.4.2	Formalization of Architecture	179
6.4.3	Formalization of Satisfies and AllocatedTo Trace Types	179
6.5	Example: Remote Patient Monitoring System	181
6.6	Generating and Validating Traces	184
6.6.1	Verification of Architecture for Functional Requirements.....	184
6.6.2	Generating Traces.....	188
6.6.3	Validating Traces.....	193

6.7	Tool Support.....	196
6.7.1	The Modeling Process.....	196
6.7.2	Tool Architecture.....	198
6.7.3	Tool Features.....	200
6.7.4	Evaluation of the Tool.....	204
6.8	Discussion on the Approach.....	209
6.9	Example for Trace Generation and Validation.....	210
6.9.1	Verification of Architecture for Functional Requirements.....	210
6.9.2	Generating Traces.....	211
6.9.3	Validating Traces.....	213
6.10	Related Work.....	217
6.10.1	Types and Semantics of Traces	217
6.10.2	Generating and Validating Traces.....	218
6.10.3	Conformance Assessment.....	219
6.10.4	Architecture Analysis.....	219
6.10.5	Analyzing AADL Models	220
6.10.6	Tool Support.....	221
6.11	Conclusions	222
7	Change Impact Analysis in Software Architecture	225
7.1	Introduction	225
7.2	Approach	227
7.3	Identifying Candidate Impacted Architectural Elements	228
7.3.1	Candidate Impacts for ‘Add Requirement’	230
7.3.2	Candidate Impacts for Other Changes	235
7.4	Proposing Architectural Changes	243
7.5	Tool Support.....	254
7.5.1	The Modeling Process.....	254
7.5.2	Tool Features.....	257
7.6	Related Work.....	260
7.6.1	Change Impact Analysis in Software Architectures.....	260

7.6.2 Tool Support.....	261
7.7 Conclusions	262
8 Conclusions	265
8.1 Introduction	265
8.2 Problems	265
8.3 Solutions.....	266
8.4 Future Research Directions	269
Samenvatting	271
REFERENCES	275
APPENDIX	289
A Definition of a model in FOL	291
B Part of the CMS Requirements Document	293
C Inference Rules in JENA	297
D Consistency Checking Rules in JENA	303
E Formal Semantics and Analysis of Behavioral AADL Models in Maude	307
F Part of the RPM Requirements Document	323
G Graphical Notation for Elements in AADL	325
H Abbreviations of Elements in the RPM System	327
I Change Impact Analysis Function for Identifying Candidate Impacted Architectural Elements	331

Table of Figures

Figure 1.1 Research Methodology	5
Figure 1.2 Within-Model and Between-Model Traces with Trace Types for Requirements and Architectural Models.....	6
Figure 1.3 Thesis Map	10
Figure 2.1 The Requirements Engineering Process [233]	15
Figure 2.2 Basic Concepts of Architecture Description (IEEE 1471 [172])	19
Figure 2.3 An Overview of the Maintenance Process [233]	25
Figure 2.4 Requirements Evolution [233]	26
Figure 2.5 Software Change Impact Analysis Process [25]	27
Figure 2.6 Core Concepts of a Tracing Approach [142]	30
Figure 2.7 Directions of Traces [142]	31
Figure 2.8 Meta-modeling Architecture	33
Figure 2.9 Transformation Pattern.....	34
Figure 3.1 Simple Directed Graph of Software Life-Cycle Objects [23]	48
Figure 3.2 Impacts Explosion without Semantics [25]	50
Figure 3.3 Requirements and Architectural Models with Traces	51
Figure 3.4 Part of Requirements and Architectural Models for Course Management System	53
Figure 3.5 Requirements and Architectural Models with Traces for Requirements Evolution	54
Figure 4.1 Within-Model and Between-Model Traces with Requirements Relation Types for Requirements and Architectural Models.....	60
Figure 4.2 Requirements Metamodel.....	62

Figure 4.3 Modeling Process with Consistency Checking and Inferencing	89
Figure 4.4 Layered Architecture of the Tool.....	90
Figure 4.5 GUI for Managing Requirements and Relations	92
Figure 4.6 Matrix View for Managing Requirements and Relations	93
Figure 4.7 Visual Editor for Managing Requirements and Relations.....	93
Figure 4.8 Output of the Inferencing Activity	94
Figure 4.9 Output of the Consistency Checking Activity	95
Figure 4.10 Explanation of the Inferred Conflicts Relation between R8 and R59	96
Figure 4.11 Explanation of the Inconsistency for R11 and R48.....	96
Figure 4.12 Explanation of the Inferred Conflicts Relation in the Inconsistency	97
Figure 4.13 Visualization of the Related Requirements for R5 with Depth 2.....	98
Figure 4.14 Example with Inferred Requires Relation	101
Figure 4.15 Analysis of the Inferred Relation to Identify Invalid Given Relations.....	102
Figure 4.16 Inconsistent Part in the Example Model	103
Figure 4.17 Analysis of the Inferred Relation in the Inconsistent Part of the Model	103
Figure 5.1 Requirements and Architectural Models Showing Within-model and Between-model Trace Relations.....	114
Figure 5.2 Wasson's Primitives for Structure of a Textual Requirement	117
Figure 5.3 Structure of a Textual Requirement based on the Definition of a Requirement in Chapter 4	118
Figure 5.4 Example Requirements Model and Traversing the Model for the Proposed Change	134
Figure 5.5 Decision Trees for the Example Requirements Model	135
Figure 5.6 Requirements Modeling Process with Change Propagation and Change Consistency Checking	145

Figure 5.7 Layered Architecture of the Tool.....	146
Figure 5.8 GUI for Proposing Changes	148
Figure 5.9 Output of the Proposing Change Activity.....	148
Figure 5.10 GUI for Propagating Proposed Changes.....	149
Figure 5.11 Matrix View for Propagating Proposed Changes.....	150
Figure 5.12 Interactive Decision Tree Builder for Propagating Proposed Changes	151
Figure 5.13 Output of the Checking Change Consistency Activity.....	152
Figure 5.14 Explanation of the Proposed Change of R16 Causing the Inconsistency	152
Figure 5.15 GUI for Implementing Proposed Changes.....	153
Figure 5.16 GUI for Implementing Propagated Proposed Changes.....	153
Figure 5.17 Output of the Impact Prediction for the Proposed Change in R7	154
Figure 5.18 Output of the Prediction Investigation for the Proposed Change in R16.....	155
Figure 5.19 GUI for the Visualization of the Propagation Paths in Impact Prediction	155
Figure 5.20 Requirements Related to R7 with Depth 2.....	157
Figure 5.21 Requirements Related to R16 with Depth 2	158
Figure 5.22 Propagation Path of the Proposed Change for R16 in the Inconsistency	159
Figure 5.23 Requirements Related to R7 with Depth 2 in IBM Rational RequisitePro.....	161
Figure 5.24 Suspended Relations for Impacted Requirements by the Change in R7.....	162
Figure 5.25 Some of the Requirements Directly/Indirectly Related to R7 in RequisitePro	164
Figure 6.1 Within-Model and Between-Model Traces with Requirements Relation Types and Trace Types between Requirements and Software Architectures	172
Figure 6.2 Overview of the Approach.....	175
Figure 6.3 Trace Metamodel for Requirements and Architecture	177
Figure 6.4 Schematic View of the Relation between P _R and P _A	180

Figure 6.5 Part of Requirements Model for RPM System	182
Figure 6.6 Overview of the RPM Architecture.....	182
Figure 6.7 Verification of Architecture for Functional Requirements.....	185
Figure 6.8 Part of the RPM Architecture.....	186
Figure 6.9 Constraints based on Semantics of Traces and Requirements Relations	190
Figure 6.10 Generated ‘Satisfies’ Trace for Requirement 5 by Using Verification Results ..	192
Figure 6.11 Venn Diagram for Generated and Actual Satisfies Traces for a Requirement .	192
Figure 6.12 Venn Diagram for Generated ‘Satisfies’ and Assigned ‘AllocatedTo’ Traces for a Requirement.....	194
Figure 6.13 Venn Diagram for Generated and Assigned ‘AllocatedTo’ Traces for a Requirement.....	195
Figure 6.14 Generated ‘Satisfies’ and Assigned ‘AllocatedTo’ Traces for Requirement 5 ...	196
Figure 6.15 Modeling Process with Trace Generation and Validation.....	197
Figure 6.16 Overview of the Tool.....	199
Figure 6.17 OSATE with AADL-Maude Plugin	201
Figure 6.18 Maude Editor in Eclipse for Verifying Architecture	202
Figure 6.19 Output of the Generating Trace Activity	203
Figure 6.20 Output of the Validating Trace Activity	204
Figure 6.21 Simulation Time as the Function of the Number of Architectural Elements...	207
Figure 6.22 Simulation Time vs. Number of States in Alloy and Maude	209
Figure 6.23 Generated ‘Satisfies’ Traces by Using Verification Results	211
Figure 6.24 Generated ‘Satisfies’ Traces by Using Requirements Relations	212
Figure 6.25 Generated Requirements Relation by Using Traces	213
Figure 6.26 Generated ‘Satisfies’ and Assigned ‘AllocatedTo’ Traces for Requirement 6 ...	214

Figure 6.27 Assigned ‘AllocatedTo’ Traces with Requirements Relation.....	214
Figure 6.28 Assigned ‘AllocatedTo’ Traces with an Invalid Requirements Relation.....	215
Figure 6.29 Given and Inferred Relations for Requirement 10	216
Figure 7.1 Within-Model and Between-Model Traces with Requirements Relation Types and Trace Types between Requirements and Software Architectures	226
Figure 7.2 Candidate Impacted Architectural Elements for the Added Requirement.....	232
Figure 7.3 Part of the RPM Architecture for Storing Blood Pressure.....	232
Figure 7.4 Changed Part of the RPM Architecture for Storing Blood Pressure	233
Figure 7.5 Part of the RPM Requirements Model.....	239
Figure 7.6 Propagation Path of the Proposed Change for Requirement 14	240
Figure 7.7 Candidate Impacted Architectural Elements for the Constraint Added to Requirement 14	241
Figure 7.8 Changed Part of the RPM Architecture for Stroring Blood Pressure.....	245
Figure 7.9 Assigned and Generated ‘AllocatedTo’ Traces for the Added Requirement	246
Figure 7.10 Last State of the Counter Example in the First Check.....	246
Figure 7.11 Last State of the Execution Trace.....	250
Figure 7.12 Another RPM Architecture for Storing CV Pressure	250
Figure 7.13 Venn Diagram for Generated and Assigned ‘AllocatedTo’ Traces for a Requirement.....	251
Figure 7.14 Requirements Modeling and Architectural Design Process with Change Impact Analysis	256
Figure 7.15 GUI for Selecting the Proposed Requirements Change	258
Figure 7.16 Output of Traversing the Propagation Path of the Proposed Requirements Change	258
Figure 7.17 Output of the Identifying Candidate Impacted Architectural Elements Activity	259

Table of Tables

Table 1.1 Mapping the Research Questions to the Chapters of the Thesis.....	11
Table 2.1 Representation of Trace Information in Traceability Approaches in MDE.....	40
Table 2.2 Mapping, Change Impact Analysis and Tool Support in Traceability Approaches	42
Table 3.1 Connectivity Matrix of Traces [23].....	48
Table 3.2 Reachability Matrix of Traces [23].....	49
Table 3.3 Some Requirements for a Course Management System	51
Table 4.1 Number of Relations and Inconsistencies in the Example	100
Table 5.1 Requirements Change Types	119
Table 5.2 Change Impact Alternatives for Domain Changes.....	136
Table 5.3 Contradicting Changes based on Semantics of Domain Changes and Change Types	140
Table 5.4 Part of Change Impact Alternatives for Our Approach and IBM Rational RequisitePro	160
Table 6.1 Confusion Matrix of Generated and Actual Traces for Satisfies Relation	193
Table 6.2 Simulation Times in the Performance Test.....	206
Table 6.3 Simulation Times in the Scalability Test.....	208
Table 7.1 Change Impact Rules for the Change Type “Add Requirement”.....	230
Table 7.2 Traversal Rules for Change Types “Delete Requirement” and “Update Requirement”.....	238
Table 7.3 Categories of the State Transition Rules in AADL with the Right-hand Side Patterns	252
Table 7.4 Right-hand Side Patterns of the State Transition Rules for Dispatching Thread T1 with Proposed Architectural Changes in AADL.....	253

Table J.1 Categories of the State Transition Rules in AADL with the Right-hand Side Patterns	337
Table J.2 Architectural Change Types	338
Table J.3 Right-hand Side Patterns of the State Transition Rules for Passing Message M1 with Proposed Architectural Changes	338
Table J.4 Right-hand Side Patterns of the Transition Rules for Dispatching Thread T1/Executing Thread T1/Switching the Mode of Thread T1 with Proposed Architectural Changes	340

Abbreviations

AADL	Architecture Analysis and Design Language
ATL	Atlas Transformation Language
CMS	Course Management System
FOL	First-Order Logic
LTL	Linear Temporal Logic
OWL	Web Ontology Language
QuadREAD	Quality-Driven Requirements Engineering and Architectural Design
R&A	Requirements & Architecture
RPM	Remote Patient Monitoring
TRIC	Tool for Requirements Inferencing and Consistency Checking

Chapter 1

1 Introduction

In this chapter, we describe the problem addressed in this thesis, together with our contributions and an outline of the thesis.

1.1 Context

At the present day, software systems get more and more complex. The requirements of software systems change continuously and new requirements emerge frequently. New and/or modified requirements are integrated with the existing ones, and adaptations to the architecture and source code of the software system are made. Integration of the new/modified requirements and adaptations to the software system are called *change management*. The size and complexity of software systems make change management costly and time consuming. 85 to 90 percent of software system budgets goes to operation and maintenance of software systems [74]. To reduce the cost of changes, it is important to apply change management as early as possible in the software development cycle. Requirements traceability is considered crucial in change management for establishing and maintaining consistency between software development artifacts. It is the ability to link requirements back to stakeholders' rationales and forward to corresponding design artifacts, code, and test cases [100]. When changes for the requirements of the software system are proposed, the impact of these changes on other requirements, design elements and source code is traced in order to determine parts of the software system to be changed. Determining the impact of changes on other parts of development artifacts is called *change impact analysis*.

This thesis is conducted within the context of the Quality-Driven Requirements Engineering and Architectural Design (QuadREAD) project [213]. The QuadREAD project aims to

bridge the gap between requirements engineer and software architect. In the project, we focus on tracing between user requirements and software architecture for change management, and in particular for change impact analysis.

In the remainder of the present chapter we introduce traceability of requirements and software architectures for change management. In the next section the problems this thesis addresses are explained. Research objective and research questions related to the problem statement are given in Section 1.3. Section 1.4 presents the research methodology that we follow in this thesis. Our solution approach is described and the contributions of the thesis are introduced in Section 1.5 and Section 1.6. Finally, we provide the outline of the thesis in Section 1.7.

1.2 Problem Statement

The need for change impact analysis is observed in both requirements and software architecture. When a change is introduced to a requirement, the requirements engineer needs to find out if any other requirement related to the changed requirement is impacted. After determining the impacted requirements, the software architect needs to identify impacted architectural elements by tracing the changed requirements to software architecture. It is hard, expensive and error prone to manually trace impacted requirements and architectural elements from the changed requirements. There are tools and approaches to automate tracing for change impact analysis like IBM Rational RequisitePro [119] and DOORS [120]. When a requirement is changed in RequisitePro, traces of the changed requirement are marked as *suspect* by the tool. RequisitePro provides two general trace types without any semantics: *traceFrom* and *traceTo*. These trace types do not say anything about the dependency between elements except the direction of the dependency. Therefore, all requirements and architectural elements directly or indirectly traced from the changed requirement (with traces marked as *suspect*) are candidate impacted. The requirements engineer has to inspect all these candidate impacted requirements and architectural elements to identify changes if there is any.

In case semantic information is missing to determine precisely how requirements and software architecture are related to each other, the requirements engineers and software architects generally have to assume the worst case dependencies based on the available syntactic information only. This generally results in a perception that a change has a wider impact on the artifacts than it is. As a result, the requirements engineers and software architects cannot precisely locate the impacted requirements and architectural elements and as such traces become useless.

Bohner [22] [23] [25] formulates the situation where all elements might be impacted, as *explosion of impacts without semantics*. He states that change impact analysis must employ additional semantic information to increase the accuracy by finding more valid impacts and excluding the invalid ones. In this thesis we tackle explosion of impacts in requirements and software architecture. Below we present an overview of the problems addressed by this thesis:

- **Explosion of Impacts in Requirements for Requirements Changes.** In practice, requirements documents are often textual artifacts with implicit structure. Most of the relations among requirements are not given explicitly. There is a lack of precise definition of relations among requirements in most tools and approaches. Due to the lack of semantics of requirements relations, change impact analysis may produce high number of false positive and false negative impacted requirements. A requirements engineer may have to analyze all requirements in the requirements document for a single change. This may result in neglecting the actual impact of a change.
- **Manual, Expensive and Error Prone Trace Establishment.** Considerable research has been devoted to relating requirements and design artifacts with source code. Less attention has been paid to relating Requirements (R) with Architecture (A) by using well-defined semantics of traces. Designing architecture based on requirements is a problem solving process that relies on human experience and creativity, and is mainly manual. The software architect may need to manually assign traces between R&A. Manual trace assignment is time-consuming, expensive and error prone. The assigned traces might be incomplete and invalid.
- **Explosion of Impacts in Software Architecture for Requirements Changes.** Due to the lack of semantics of traces between R&A, change impact analysis may produce high number of false positive and false negative impacted architectural elements. A software architect may have to analyze all architectural elements in the software architecture for a single requirements change.

1.3 Research Questions

The objective of this thesis is to investigate to what extent and how traceability can be used to support change management for requirements and software architecture by enhancing traces with semantics. Within the context of this objective, we provide a traceability framework of requirements and software architectures for change management. A number

of research questions need to be answered. Answering these questions will give us a better understanding of the problem domain and the deficiencies of the current solutions.

- *Research Question 1:* What does traceability mean? Can every relation between software development artifacts or between elements in the artifacts be a trace? What is the criterion for a relation to be a trace?
- *Research Question 2:* What are the current traceability approaches for change management? What are their deficiencies? Which solutions and technologies have been proposed to address these deficiencies?
- *Research Question 3:* What are the change scenarios for requirements and software architecture? What is necessary for these change scenarios to be handled? Which solutions can be used?
- *Research Question 4:* How to model requirements, software architecture and traces with their semantics for change management? What aspects of requirements, software architecture and traces should be modeled and how? How can we use the modeled aspects to reason about requirements, software architecture and traces?
- *Research Question 5:* How can a change in a requirement be propagated to other requirements and to software architecture? How can we support the requirements engineer and software architect for performing changes? How can we formally check if the evolved architecture satisfies evolved requirements? How can we become sure that traces are up-to-date?

These questions guide the research presented in this thesis. In Section 1.7, we give the outline of the thesis and a table that relates the research questions to the chapters in which we provide answers to the questions (see Table 1.1).

1.4 Research Methodology

In this thesis, we try to solve two types of problems: *knowledge problems* and *design problems* [259] [260] [261]. The difference between the current and desired knowledge states is the knowledge problem. The difference between the current and desired state of the world is a design problem.

Our research methodology has three phases – *problem analysis*, *solution design* and *solution validation* (see Figure 1.1).

In the first phase, we solve a knowledge problem, for instance, we want to understand what traceability means, what the criterion for a relation to be a trace is, what the current traceability approaches for change management are, and what the deficiencies of the current approaches are. For that purpose, we analyze the literature about traceability from different research areas (*Change Management, Model Driven Engineering and Requirements Engineering*) to discover possible change management problems in current traceability approaches for requirements and software architectures.

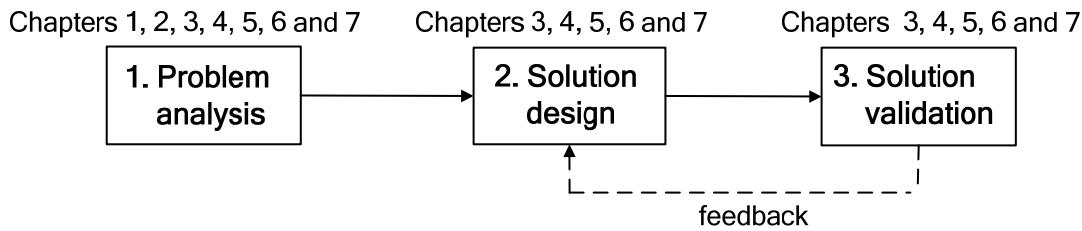


Figure 1.1 Research Methodology

In the second phase, the results of the first phase are used to design a new solution. We solve a design problem, that is, we provide a traceability framework of requirements and software architectures for change management. Our goal is to improve change management for requirements and software architectures by providing semantics of traces.

Finally, in the third phase, we validate our solution by investigating its availability for the problems discovered in the problem analysis phase. This is a knowledge problem since we want to gain knowledge about the properties of our solution, and the relation between the solution and the problems. The outcome of the solution validation phase is fed back to the solution design phase in order to improve the solution.

1.5 Approach

We choose Model Driven Engineering (MDE) as a solution platform for our approach. MDE provides a uniform treatment of software artifacts, such as requirements documents, software design and test documents, as models. It also enables using different formalisms to reason about development artifacts described as models. To give an explicit structure to requirements documents and treat requirements, architecture and traces in a uniform way, we use metamodels and models within the context of MDE. Figure 1.2 gives requirements model, architecture model and traces between Requirements (R) and Architecture (A). Traces between R&A are also described as a model although the trace model is not explicitly shown in Figure 1.2.

To cope with the problem of impact explosion in requirements and software architectures due to the lack of semantic information, we study modeling of requirements, software architectures and traces with semantic information. We distinguish types of traces between requirements, and between requirements and software architectures (see Figure 1.2). We provide a traceability approach for change management for requirements and software architectures by using semantics of traces.

To provide an explicit structure to requirements documents, we present a requirements metamodel with most commonly found entities in literature. The most important elements of the requirements metamodel are requirements relations and their types. We give formal requirements relation types to be able to reason about requirements and their relations.

To be able to use the semantics of requirements relations for change impact analysis in requirements, we give a classification of requirements changes based on the structure of a textual requirement provided with formal semantics. The formalization of requirements relations and changes are used in order to overcome the explosion of impacts in requirements.

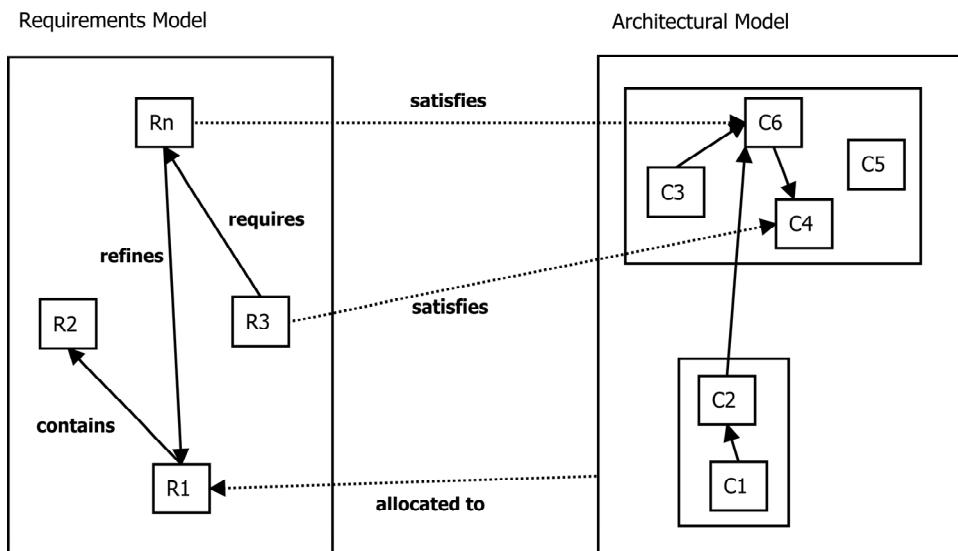


Figure 1.2 Within-Model and Between-Model Traces¹ with Trace Types for Requirements and Architectural Models

For the evolution of requirements, we provide techniques for analyzing the impact of requirements changes on architecture design. We need techniques for trace establishment between R&A. Therefore, we give trace types and their semantics in order to link

¹ See [142] For the terminology of Within-Model and Between-Model traces

requirements to software architectures in a similar way used for requirements relations. Requirements relations and architecture verification techniques are used for trace establishment between R&A.

In order to perform change impact analysis in software architectures, we combine the architecture verification techniques and their output (counter example) with the use of semantics of traces between R&A.

1.6 Contributions

This thesis provides the following contributions:

- A modeling language for definition of requirements models with formal semantics

Chapter 4 presents a modeling language for definition of requirements models with formal semantics. The language is defined according to the MDE principles by defining a metamodel. It is based on a survey about the most commonly found requirements types and relation types. With this language, the requirements engineer can explicitly specify the requirements and the relations among them. The semantics of these entities is given in First Order Logic (FOL) and allows two activities. First, new relations among requirements can be inferred from the initial set of relations. Second, requirements models can be automatically checked for consistency of the relations. Tool for Requirements Inferencing and Consistency Checking (TRIC) is developed to support both activities. The defined semantics is used in a technique for change impact analysis in requirements models.

- A change impact analysis technique for requirements using semantics of requirements relations and requirements change types

Chapter 5 addresses the problem of explosion of impacts in requirements when semantics of requirements relations is missing. The technique uses formal semantics of requirements relations and requirements change types. A classification of requirements changes based on the structure of a textual requirement is given and formalized. The semantics of requirements change types is based on FOL. We support three activities for impact analysis. First, the requirements engineer proposes changes according to the change classification before implementing the actual changes. Second, the requirements engineer identifies the propagation of the changes to related requirements. The change alternatives in the propagation are determined based on the semantics of change types and requirements relations. Third, possible contradicting changes are identified. We extend TRIC with a support for these activities. The tool automatically determines the change propagation paths,

checks the consistency of the changes, and suggests alternatives for implementing the change. With change alternatives and propagation paths we eliminate some false positive impacted requirements. We provide a more precise change impact analysis in requirements models than requirements management tools like RequisitePro.

- A technique that provides trace establishment between R&A by using architecture verification and semantics of traces

Chapter 6 presents an approach that provides trace establishment by using architecture verification together with semantics of requirements relations and traces. We use a trace metamodel with commonly used trace types. The semantics of traces is formalized in FOL. Software architectures are expressed in the Architecture Analysis and Design Language (AADL) [225]. AADL is provided with a formal semantics expressed in Maude [198] [197]. The Maude tool set allows simulation and verification of architectures. The first way to establish traces is to use architecture verification techniques. A given requirement is reformulated as a property in terms of the architecture. The architecture is executed and a state space is produced. This execution simulates the behavior of the system on the architectural level. The property derived from the requirement is checked by the Maude model checker. Traces are generated between the requirement and the architectural components used in the verification of the property. The second way to establish traces is to use the requirements relations together with the semantics of traces. Requirements relations are reflected in the connections among the traced architectural elements based on the semantics of traces. Therefore, new traces are inferred from existing traces by using requirements relations. We use semantics of requirements relations and traces to both generate/validate traces and generate/validate requirements relations. There is a tool support for our approach. The tool provides the following: (1) generation/validation of traces by using requirements relations and/or verification of architecture, (2) generation/validation of requirements relations by using traces. We improve trace establishment between R&A with automation and trace validation.

- A change impact analysis technique for software architecture using architecture verification and semantics of traces between R&A

Chapter 7 presents a change impact analysis technique for software architecture using architecture verification and semantics of traces. The technique is semi-automatic and requires participation of the software architect. Our technique has two parts. The first part is to identify the architectural elements that implement the system properties to which proposed requirements changes are introduced. By having the formal semantics of

requirements relations and traces, we identify which parts of software architecture are impacted by a proposed change in requirements. We eliminate some false positive impacted architectural elements. We have extended TRIC for determining candidate impacted architectural elements. The second part of our technique is to propose possible changes for software architecture when the software architecture does not satisfy the new and/or changed requirements. The technique is based on architecture verification. The output of verification is a counter example if the requirements are not satisfied. The counter example is used with a classification of architectural changes in order to propose changes in the software architecture. These changes produce a new version of the architecture that possibly satisfies the new or the changed requirements. By eliminating some false positive impacts and proposing architectural changes, we provide a more precise change impact analysis in software architecture than requirements management tools like RequisitePro.

1.7 Outline of the Thesis

Figure 1.3 shows the map of the thesis with chapters and relations among them.

The thesis consists of the following chapters:

Chapter 2 Background and Definitions. This chapter describes the concepts used in the thesis. It introduces concepts and techniques from the areas of Requirements Engineering, Software Architecture, Traceability, Software Change Management and Model Driven Engineering as they are described in literature. Furthermore, the literature survey for existing traceability approaches in MDE is given in general and also in particular for change impact analysis. The literature survey is based on work published in [87].

Chapter 3 Analysis of Impacts Explosion in Traceability. This chapter motivates the need for semantics of traces between requirements, and requirements & architecture for change management by addressing the impacts explosion problem with some change scenarios.

Chapter 4 Semantics of Requirements Relations. This chapter studies formal definitions of requirements relation types in order to enable reasoning about requirements relations. The requirements metamodel with commonly used relation types and their semantics are given in this chapter. The features of TRIC for requirements inferencing and consistency checking are presented. We illustrate our approach in an example which shows that the formal semantics of relation types enables new relations to be inferred and contradicting relations in requirements documents to be determined. This chapter is based on work published in [96] and [98].

Chapter 5 Change Impact Analysis in Requirements. This chapter discusses problems related to change impact analysis in requirements and provides the approach for the discussed problems by using formal semantics of requirements relations in Chapter 4 and requirements change types. The features of TRIC for change impact analysis are presented. We illustrate our approach in an example which shows that the formal semantics of relation types and change types enables proposed changes to be propagated and contradicting proposed changes in requirements to be determined. This chapter is an enhancement of results published in [95] and [243].

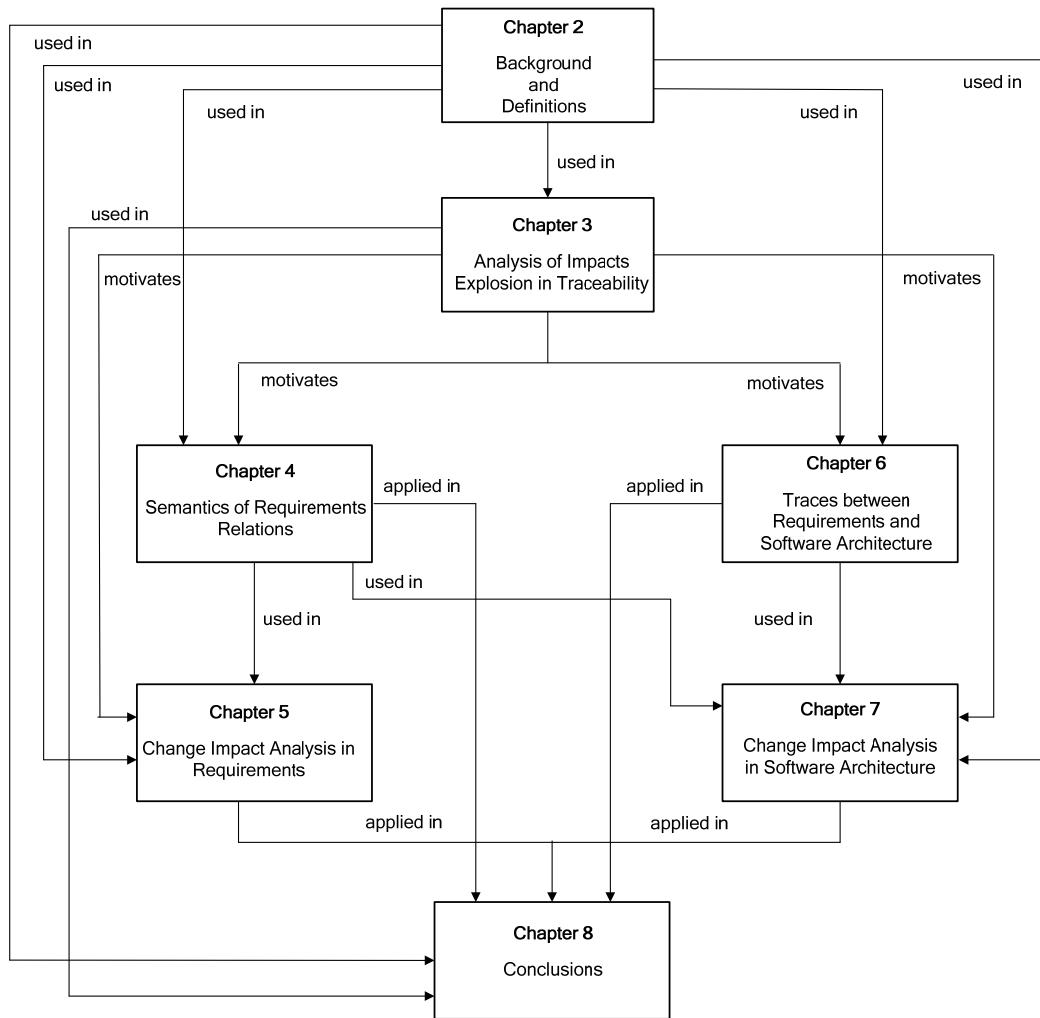


Figure 1.3 Thesis Map

Chapter 6 Traces between Requirements and Software Architecture. This chapter presents the approach that provides trace establishment by using semantics of traces between Requirements (R) and Architecture (A). Requirements relations and architecture

verification techniques are used in the approach. The trace metamodel with commonly used trace types are presented in this chapter. The tool support for trace establishment is presented. We illustrate our approach in an example which shows that the formal semantics of trace types with architecture verification techniques enables traces between R&A to be generated and contradicting traces to be determined. This chapter is an enhancement of results published in [97].

Chapter 7 Change Impact Analysis in Software Architecture. This chapter presents the approach on how to perform change impact analysis in software architectures by using architecture verification techniques and traces between R&A. The tool support for change impact analysis in software architectures is presented. We illustrate our approach in an example which shows that the formal semantics of trace types with architecture verification techniques enables impacted architectural elements for requirements changes to be determined.

Chapter 8 Conclusions. This chapter gives conclusions and an evaluation of the contributions in this thesis, and describes directions for future work.

Table 1.1 relates the research questions to the chapters in which we provide answers to the questions.

Table 1.1 Mapping the Research Questions to the Chapters of the Thesis

		Chapter							
		1	2	3	4	5	6	7	8
Research Question 1		+							
		+							
			+						
				+			+	+	
						+	+	+	

Chapter 2

2 Background and Definitions

In our work, we utilize concepts and techniques from the areas of requirements engineering, software architectures, software change management, traceability and Model Driven Engineering (MDE). In this chapter, we provide background information on these areas and introduce a set of definitions used throughout the thesis.

2.1 Introduction

This chapter gives an overview of basic concepts used in the thesis. Various definitions of these concepts are found in literature. We aim at selecting a consistent set of definitions that support the understanding of the thesis.

In this chapter we answer Research Question 1 (*What does traceability mean? Can every relation between software development artifacts or between elements in the artifacts be a trace? What is the criterion for a relation to be a trace?*) and Research Question 2 (*What are the current traceability approaches for change management? What are their deficiencies? Which solutions and technologies have been proposed to address these deficiencies?*) raised in Chapter 1. With the definitions in this chapter we explain traceability within the context of change management for requirements and software architecture. This chapter also presents a survey of traceability techniques in MDE in which we study the current approaches for traceability with their deficiencies.

The structure of the chapter is as follows. Section 2.2 describes Requirements Engineering by mentioning about fundamentals of Requirements Engineering such as requirements engineering process and requirements documentation. Section 2.3 gives the basic concepts of software architecture design and analysis. Section 2.4 gives the details of software change management. Section 2.5 discusses definitions of trace with traceability types. Section 2.6

describes the notion of Model Driven Engineering (MDE) as an enhancement of Model Driven Architecture (MDA). Section 2.7 discusses the state-of-the-art in traceability approaches in MDE.

2.2 Requirements Engineering

Requirements engineering is the process of finding out, analyzing, documenting and checking the services and constraints for the system to be built [233]. Requirements are the descriptions of these services and constraints for the system. Van Lamsweerde [151] describes requirements engineering as “a coordinated set of activities for exploring, evaluating, documenting, consolidating, revising and adapting the objectives, capabilities, qualities, constraints and assumptions that the system-to-be should meet based on problems raised by the system-as-is and opportunities provided by new technologies”.

In this section, we begin with presentation of the key terms of the field of requirements engineering and concepts related to these terms. The definitions of software requirement are explored. We introduce the requirements engineering process. Then, approaches for software requirements specification and documentation are presented.

2.2.1 Software Requirements

There are a number of definitions and classifications of requirement in literature. Sommerville [233] defines a requirement as “the descriptions of the services and constraints for the system”. In SWEBook [239], a property which must be exhibited by a system is called a requirement. We use this definition as our working definition for requirements in the thesis.

Different terms such as user requirements, system requirements, software requirements functional/non-functional requirements and quality requirements are used to classify requirements in literature. For instance, user requirements mean the high-level abstract requirements and system requirements mean the detailed description of what the system should do [233]. We consider software requirements or software system requirements synonyms and as a specialization of system requirements for software systems in this thesis. Software requirements are often classified as functional, non-functional and domain requirements [233].

- **Functional Requirements.** The functional requirements for a system describe the functionality or services that the system is expected to provide.

- **Non-Functional Requirements.** These requirements are related to emergent system properties such as reliability, performance or adaptability, or they define constraints on the system such as capabilities of I/O devices.
- **Domain Requirements.** These requirements are derived from the application domain of the system. They might be functional or non-functional.

2.2.2 Requirements Engineering Processes

Requirements engineering can be described as a process of a collection of activities to create and maintain a requirements document [233]. There are four activities involved in requirements engineering process [233]: (a) feasibility study, (b) requirements elicitation and analysis, (c) requirements specification and documentation, and (d) requirements validation (see Figure 2.1). In addition to these activities, there is an additional requirements engineering activity not listed in Figure 2.1, which is requirements management. Requirements management is concerned with managing requirements change.

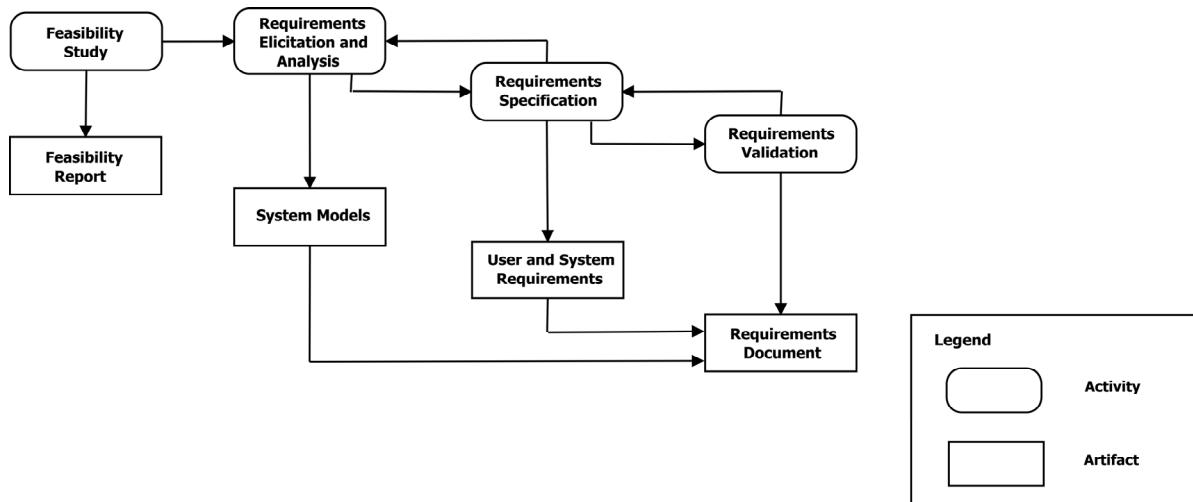


Figure 2.1 The Requirements Engineering Process [233]

- **Feasibility Study.** A feasibility study investigates the system contribution to the organization objectives, integration of the system with current systems, and the feasibility of the implementation of the system by using current technology within given constraints [233]. The outline of the system is the input of the feasibility study and the output is a report which recommends whether or not it is worth carrying on with the requirements engineering and system development process.
- **Requirements Elicitation and Analysis.** Requirements engineers and software architects work with system stakeholders and end-users to find out about the

application domain, what services the system should provide, the required performance of the system, hardware constraints and so on [233]. The output of requirements elicitation and analysis is general objectives, system requirements, software requirements, user requirements, relevant domain properties and concept definitions.

- **Requirements Specification.** The results of the elicitation and analysis activity need to be precisely defined and documented. The output of the requirements specification activity is the first version of the requirements document. Requirements specification provides an assessment of requirements with a basis for estimating product costs, risks, and schedules before design begins [239].
- **Requirements Validation.** Requirements are checked if they actually define the system which customer wants. Requirements validation has much in common with requirements elicitation and analysis but they are distinct since requirements validation is concerned with complete version of the requirements document whereas elicitation and analysis works on incomplete requirements.
- **Requirements Management.** The requirements of software systems are mostly changing in time. Requirements management is about understanding and controlling changes to system requirements. The input of the requirements management is the changes in understanding of the system to be built and the output is the revised requirements in the requirements document. Requirements management has itself sub-activities: problem analysis and change specification, change analysis and costing, and change implementation.

In this thesis, we provide techniques and tools for change management in requirements and software architecture. Our work mostly supports the requirements management activity.

2.2.3 Software Requirements Specification and Documentation

Software requirements specification is an agreement among stakeholders of the system on what the software system is to do, as well as what it is not expected to do.

For non-technical readers, the software requirements specification document is often accompanied by a software requirements definition document. Software requirements are often written in natural language in requirements document, but this may be supplemented by formal or semi-formal descriptions [239]. Requirements specification and documentation techniques are the following [151]: (a) Documentation in Natural Language, (b) Use of Diagrammatic Notations, and (c) Formal Specification.

- **Documentation in Natural Language.** Agreed statements in requirements elicitation and analysis can be documents in natural language. The first option is to see free documentation in unrestricted natural language. There are no limitations in expressiveness on what requirements engineer can specify in natural language whereas unrestricted use of natural language might cause ambiguities, forward references, unmeasurable statements and opacity in requirements document [151]. Disciplined documentation in structured natural language can be used to overcome these defects in requirements documents. Use of predefined statements templates, requirements document templates, and decision tables are examples of disciplined documentation in structured natural language.
- **Use of Diagrammatic Notations.** Semi-formal specification languages can be used to complement the use of natural language. Here, semi-formal means that the entities in requirements document and their relations are declared in some machine-processable form with well-defined language syntax whereas the statements about these entities are informally specified in natural language [151]. The use of context, problem, frame, dataflow, use-case and entity-relationship diagrams is example of the use of diagrammatic notations.
- **Formal Specification.** Formal specification provides the formalization of statements which are left informal in the use of diagrammatic notations. The benefits of the use of formal specification is high degree of precision of requirements, precise rules for interpretation of requirements and sophisticated forms of validation and verification of requirements that can be automated by tools [151]. On the other hand, formal specification requires knowledge on formal methods and high effort for the formulation of requirements from requirements engineers.

In this thesis, requirements and their relations are defined by using a requirements metamodel. In the requirements metamodel, requirements are captured in a requirements model. A requirements model contains requirements and their relationships. The descriptions of requirements are informally specified in natural language. The approach that we follow for documentation of requirements in this thesis can be considered as the use of diagrammatic notations.

2.3 Software Architecture

We begin presentation of the key terms in the field of software architecture and concepts related to these terms. The definitions of software architecture are explored. We give major

constituent elements of architectures, including architecture patterns and styles. Then, approaches for modeling software architectures are presented.

2.3.1 Definitions of Software Architecture

Architecture is a popular term in the computing community and it is used in various contexts to mean the software components in a software system, the structure of the central processing unit, or the organizational structure of the information systems. There are also different interpretations and definitions of the term architecture within the context of software components of a software system.

Bas et al. [16] define software architecture as: “the structure or structures of a program or computing system, which comprise elements, the properties of those elements, and the relationships among them”.

Perry and Wolf [210] formulate the definition of software architecture as a triple where $Software\ Architecture = \{elements, form, rationale\}$. Similar to Bas et al., Perry and Wolf considers the architecture as the systems’ key elements, and their relationships to each other and to their environment. Elements are the system’s building blocks where the form is the organization of system elements in the architecture. Rationale captures the software architect’s intent, assumptions, decisions, and constraints effecting the architect’s decisions for the architecture. Different than Bas et al., Perry and Wolf explicitly consider the rationale of the software architect in the definition of software architecture.

Another definition of software architecture which is mainly about design decisions which is part of rationale is given by Taylor et al. [242]. Taylor et al. define the architecture as: “the set of principal design decisions made about the system”. The notion of design decision is central to software architecture and to all of the concepts based on it [242].

Klusener et al. [141] provide a different perspective on how to define a software architecture. They define software architecture in the following: “the software architecture of deployed software is determined by those aspects that are the hardest to change”.

One more definition of software architecture is from the IEEE 1471 standard [172] which is a recommended practice for architectural description of software-intensive systems. According to IEEE 1471 standard, architecture is a fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution. Like Perry et al. and Bas et al., IEEE 1471 standard considers the elements of the system and their relations among the elements as architecture. However, the definition in IEEE 1471 standard does not specifically refer to

software. IEEE 1471 standard introduces basic concepts in software architecture and relationships among them within the context of software architecture description (see Figure 2.2). An architecture description is a collection of documents to describe a system's architecture [172].

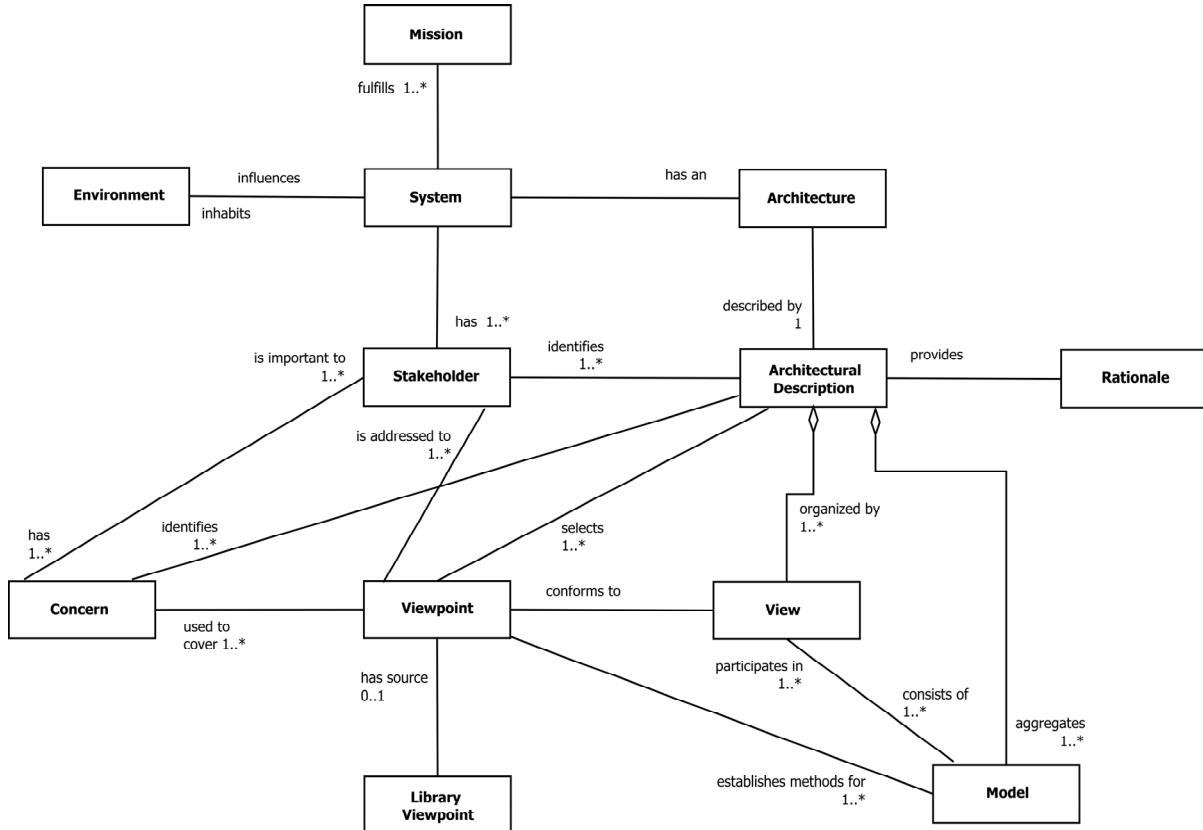


Figure 2.2 Basic Concepts of Architecture Description (IEEE 1471 [172])

An individual, team, or organization with interests in, or concerns relative to, a system are called *stakeholders*. These might include end users, operators, software architects, developers, subcontractors, and maintainers. A *concern* is a stakeholder's interest which pertains to the development, operation, or other key characteristics of the system such as run-time behavior, performance, reliability, security, evolvability, or distribution. Stakeholders may have different and possibly conflicting concerns. A *view* is a representation of the whole system from the perspective of a related set of concerns. The architectural views are the actual description of the system. A *viewpoint* determines the resources and rules for constructing a view.

2.3.2 Software Architecture Analysis

Software architecture is one of the important artifacts of software development since it enables reasoning about the system by capturing early design decisions. Therefore, it is important that software architecture reflect a certain abstraction of the system which enables to focus on the relevant parts of the system for analysis. Software architecture analysis helps reducing unnecessary maintenance costs by providing reasoning about the system before it is built. Software architecture analysis techniques are divided into three categories [242]: (a) inspection- and review- based, (b) model-based, and (c) simulation-based.

- **Inspections and Reviews.** They are manual analysis techniques used by different stakeholders to ensure a variety of properties in a software architecture such as scalability, or adaptability. Since these techniques are manual, they are very human intensive and they can be very costly. On the other hand, they have the advantage of being useful in the case of informal or partial architectural descriptions [242]. Examples of inspection- and review-based methods are the Architectural Trade-Off Analysis Method (ATAM) and the Scenario-based Architecture Analysis Method (SAAM) [53].
- **Model-Based Analysis.** It is usually automatic where models are used to analyze system properties in architectural level such as structural properties, behavioral properties and non-functional properties. Compared to inspection and reviews, model-based techniques are less human intensive and less costly. However, they can only be used to analyze properties which can be encoded in the architectural model [242]. They are not for implicit properties which are inferred by human from the non modeled existing information. Architecture description languages such as Wright [9], Aesop [89], and MetaH [115] support model-based analysis.
- **Simulation-Based Analysis.** It is used to analyze the behavior of software architectures by using an executable architecture model of a given system. The results of the simulation can be manually or automatically inspected. Since software architecture abstracts some details of the system, simulation of the architecture may not produce identical results to the system's execution. The output of simulation might be observed only for event sequences, general trends, or range of values rather than specific results [242]. An example simulation analysis platform is the eXtensible Tool-chain for Evaluation of Architectural Models (XTEAM) [68] which is a model driven architectural description and simulation environment for mobile software systems. Not all architectural models are available for simulation-based analysis. Available architectural models mostly need to be mapped to an external formalism

such as discrete event system simulation formalism or queueing network in order to enable simulation.

In this thesis, we map AADL architecture models to rewriting logic [48] [49] in order to perform simulation-based analysis.

2.3.3 Architectural Patterns and Styles

An architectural pattern is a description of an element and relation types together with a set of constraints on how they may be used [16]. Similar to design pattern [88], an architectural pattern provides a common vocabulary to build an architecture. This common vocabulary is used for communication between stakeholders and software architects. A synonym for architectural pattern is architectural style. A pattern restricts many of the possible design choices and prevents possible design errors in the architecture. Examples of architectural patterns are clients and servers, pipes and filters, and layered architectural patterns. For instance, layered architectural pattern [42] restricts a system with two or more layers stacked upon each other. A layer n is only allowed to communicate with the layers it has direct contact with.

2.3.4 Modeling Software Architecture

As stated in definitions of software architecture in Section 2.3.1, software architecture can be considered as the set of design decisions made about a software system. An architectural model is an artifact that captures some or all of the design decisions that comprise a system's architecture [242]. Architecture modeling is the reification and documentation of those design decisions. In the thesis, architectural models are our primary interest. Taylor et al. [242] classifies architecture modeling techniques as following: (a) generic techniques, (b) early architecture description techniques, (c) domain- and style-specific ADLs, and (d) extensible ADLs.

2.3.4.1 Generic Techniques

These techniques are not specifically developed to describe a software architecture. Natural language, informal graphical PowerPoint-style modeling and the Unified Modeling Language (UML) are considered as generic techniques for modeling software architecture. Natural languages are expressive but they are ambiguous and nonrigorous since they have imprecise semantics about software architecture [242]. They can only be checked by humans. Ambiguity problems in natural languages can be limited by using a restricted form of natural languages with consistent dictionary of software architecture terms. Tools like Microsoft PowerPoint provide users graphical diagrams to model software architectures. These diagrams are good to capture early ideas but it is difficult to interpret their meaning since

they have imprecise semantics. UML is more precise than arbitrary diagrams that would be produced in PowerPoint [242]. However, most constructs in UML are still semantically ambiguous. Stakeholders should make agreements about how to interpret UML diagrams in order to model software architecture in UML. Stereotypes, tagged values and Object Constraint Language (OCL) can be used to extend UML for architecture modeling purposes such as avoiding ambiguities in software architecture models.

2.3.4.2 Early Architecture Description Techniques

The research in 1990s on how to best capture software architectures result in architecture description languages (ADLs) developed specifically for modeling software architecture. Medvidovic and Taylor [175] survey early ADLs and provide a classification framework for these ADLs consisting of the following four common architectural elements: *components*, *connectors*, *interfaces* and *configurations*.

Apart from generic techniques, these architecture description languages are semantically precise but they are not flexible. Examples of first generation languages and their scopes are as follows [242]:

Darwin [169]. It is used to model architectures of highly-distributed systems whose dynamism is guided by strict formal underpinnings.

Rapide [165]. It is used for modeling and simulation of dynamic behavior of software architecture.

Wright [9]. It is for modeling and analysis (specifically deadlock analysis) of dynamic behavior of concurrent systems.

2.3.4.3 Domain- and Style-Specific ADLs

Early architecture description languages are used to model a wide variety of software systems. However, they can not be tailored to stakeholder needs since they do not target a particular group of stakeholders. Domain- and style specific ADLs are proposed to avoid this kind of problems encountered in early ADLs. Examples of domain- and style- specific ADLs are as follows [242]:

Koala [203]. It was developed by Philips Electronics to model the architecture of consumer electronics devices such as televisions and DVD players.

Weaves [99]. It is used to model data-flows characterized by high-volume of data and real-time requirements.

The Architecture Analysis and Design Language (AADL) [225] [79]. It is developed by Software Engineering Institute in Carnegie Mellon University to specify system architectures for a wide variety of embedded and real-time systems such as automotive, avionics and medical systems. In this thesis, we use AADL to specify software architectures.

2.3.4.4 Extensible ADLs

Extensible ADLs are proposed to combine the flexibility of early ADLs and expressivity of domain- and style-specific languages with the analyzability and precision of semantically rich languages. These languages provide basic constructs for describing common architectural elements with extending these elements for user-defined constructs [242]. Examples of extensible ADLs are as follows [242]:

Acme [90]. It is designed to be an interchange language for several existing ADLs. It has a base of constructs to be extended: components, connectors, ports, roles, attachments, systems, and representations.

The Architecture Description Markup Language (ADML) [234]. It is an XML based ADL. It provides meta-properties which are used to specify user-defined properties and property types.

xADL [60]. It is build upon XML and schemas. The default schema provides the basic elements: component, connector, interface and configuration. The default schema is extended for modeling different types of systems.

In this thesis, we use the definition of software architecture by Bas et al. [16]. Software architectures are expressed in Architecture Analysis and Design Language (AADL). We use formal dynamic semantics for part of AADL given in rewriting logic used in Maude language and tools. Formal semantics of AADL enables performing simulation and verification of AADL models (simulation-based analysis). It is used to analyze the behavior of software architectures by using an executable architecture model of a given system.

2.4 Software Change Management

In this section we first introduce the strategies for software change management. Software maintenance, one of the strategies for change management, is explored in detail. We then give the details of requirements evolution within the context of software maintenance. In the end, change impact analysis, an activity in software maintenance, is introduced since our focus in the thesis is to determine the impacted requirements and architectural elements in response to changes in requirements of the software system.

2.4.1 Strategies for Software Change Management

Generally, systems of any size need to be changed. Changes might happen in different contexts such as new requirements may emerge, existing requirements might change, coding and design errors might arise. Different changes require different strategies to be handled. Types of strategies for software change management are the following [257]:

- **Software Maintenance.** This is the strategy for handling changes in a software system after the system has been put into use [233]. Changes within the context of software maintenance are fixing coding and design errors, adapting software systems for a new operating system, and modifying system functionality in response to changes in organizational or business needs.
- **Architectural Transformation.** It is the software change strategy for significant changes in the architecture of the software system. An example is that the software system evolves from a client-server architecture to a broker architecture. There might be different reasons for architectural changes like hardware costs, user interface expectations and distributed access to systems [233].
- **Software Re-engineering.** This strategy involves changes made to make the software system easier to understand and improve the quality of the software system. The software re-engineering strategy consists of the activities *source code translation*, *reverse engineering*, *program structure improvement*, *program modularization* and *data reengineering*.

These strategies are not mutually exclusive [233]. Software re-engineering could be performed before architectural transformation in order to make software system easier to understand for architectural changes. The thesis focuses on modifying system functionality in response to changes in organizational or business needs. Therefore, we give more details about software maintenance in the next section.

2.4.2 Software Maintenance

Software maintenance is the process of changing a software system after it has been put into use [233]. Changes within the context of software maintenance could be correcting coding errors, correcting design errors, correcting requirements errors, simply implementing new system features or modifying existing system features. Types of software maintenance are the following [233]:

- **Maintenance to Repair Software Faults.** This type of maintenance is to repair coding, design and requirements errors.

- **Maintenance to Adapt the Software to a Different Operating System.** This type of maintenance is necessary to adapt the software system to cope with changes in hardware, operating system or other supporting software.
- **Maintenance to Add to or Modify the System's Functionality.** This type of maintenance is required to add or modify system features in case of changes in business and organizational needs, which cause changes in software system requirements.

Swanson [237] addresses the types of maintenance as *corrective*, *adaptive* and *perfective* maintenance. Corrective maintenance is performed in response to processing, performance and implementation failures. Changes in data environment or in processing environment cause adaptive maintenance. Perfective maintenance is performed to enhance performance, or improve maintainability [237].

The maintenance has activities such as *change impact analysis*, *release planning*, and *change implementation*. Change requests from stakeholders of the system, such as system users, project managers or programmers, trigger the maintenance. The cost and impact of the changes are assessed in change impact analysis to see which part of the system is affected and to estimate the cost of changes [233].

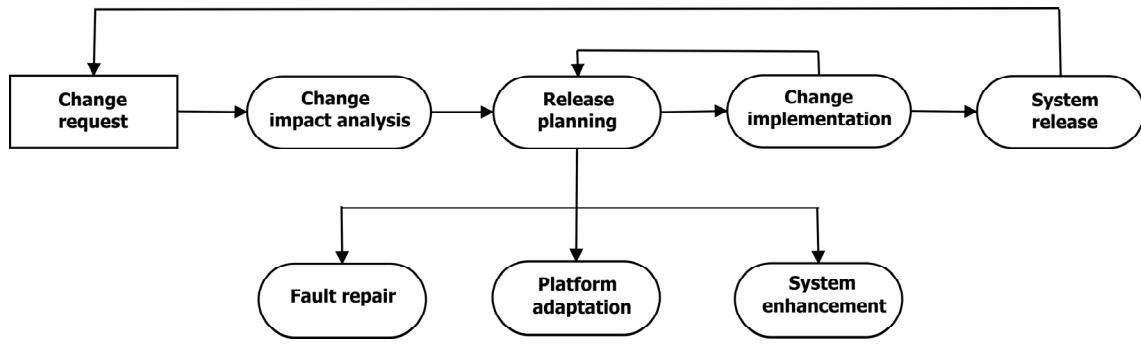


Figure 2.3 An Overview of the Maintenance Process [233]

Based on the output of change impact analysis, changes to implement for the next release of the software system are described in the release planning. Finally, the decided changes are implemented during change implementation.

This thesis addresses the type of maintenance to add to or to modify the system's functionality where requirements of the system evolve in case of changes in business and organizational needs. We do not address cost estimation, release planning or implementation. In the thesis, we focus only on change impact analysis.

2.4.3 Requirements Evolution

Requirements evolve during the development life cycle as a result of changes in business and organizational needs (see Figure 2.4). If evolution of requirements is not managed properly, there might be requirements that are not implemented as they are described in the final release of the software system. This increases the cost of software system and leads to invalid systems.

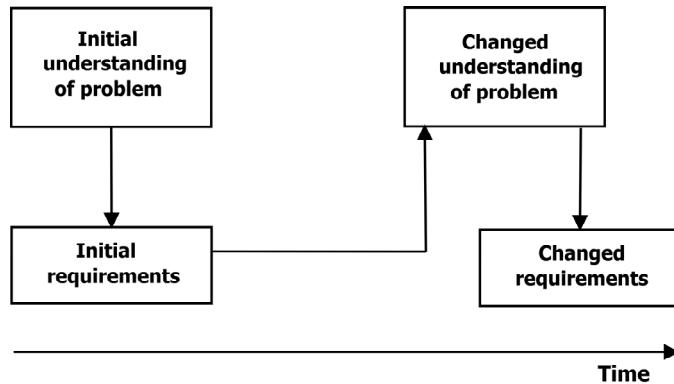


Figure 2.4 Requirements Evolution [233]

From requirements evolution perspective, requirements are classified as *enduring requirements* and *volatile requirements* [233].

- **Enduring Requirements.** They are core requirements about the domain of the system, such as requirements about students, lecturers for a course management system.
- **Volatile Requirements.** They are requirements changing while the system is being developed or after the system has been put into operations. For example, requirements about student registration regulation, which depend on yearly school policies, are volatile requirements.

In the thesis, we address volatile requirements within the scope of software maintenance and in particular, in change impact analysis. New requirements may emerge or existing requirements might change. Techniques for the analysis of the impact of these requirements changes on other requirements and architectural elements are developed in this thesis.

2.4.4 Change Impact Analysis

Change impact analysis is defined as “identifying the potential consequences of a change, or estimating what needs to be modified to accomplish a change” [24] [23] [22] [25]. The first part of the definition, identifying the potential consequences of a change, addresses research

issues such as predicting the effort required to modify work products [196]. The second part of the definition addresses analyzing source code dependencies and traces between development artifacts to determine impacted elements. This thesis focuses on analyzing traces between requirements and between requirements & software architecture in order to determine the impacted parts of requirements documents and software architecture for requirements changes. Figure 2.5 depicts the software change impact analysis as a process.

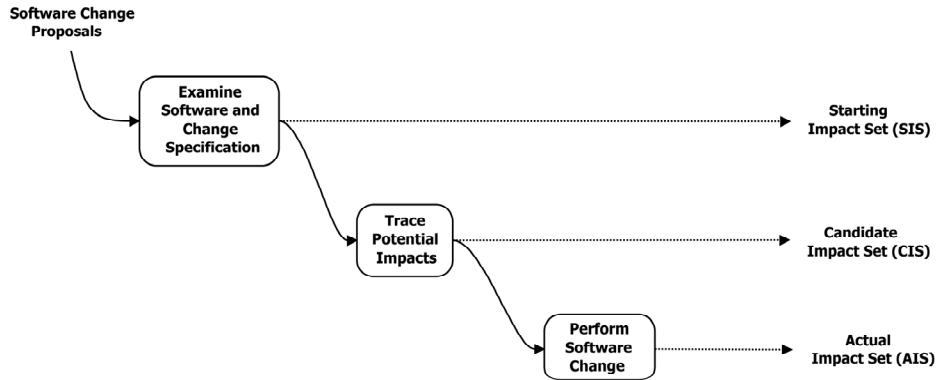


Figure 2.5 Software Change Impact Analysis Process [25]

The process for change impact analysis is iterative. The requirements engineer or software architect receives the software change proposals as a change specification from the stakeholders of the software system. The change specification contains a series of change requests for the software system. Software system and change specification are examined in order to determine the starting impact set (SIS). The SIS is the initial set of elements thought to be affected by a change. After tracing the potential impacts, the set of elements to be affected is estimated (the candidate impact set - CIS). The actual impact set (AIS) is the set of elements actually modified. There might be more impacts discovered (the discovered impact set – DIS) during performing software change. The false-positive impact set (FPIS) is the set of over-estimate of impacts. Then, we have $AIS = CIS + DIS - FPIS$.

There are change impact analysis techniques that determine the sets of impacted elements. Bohner and Arnold [24] describe two types of change impact analysis techniques, *traceability* and *dependency change impact analysis*. Kilpinen [139] describes a third type, *experimental impact analysis*.

- **Traceability Impact Analysis.** In traceability impact analysis, traces between requirements, software design, source code and tests, which are the main artifacts of software development life-cycle, are used in order to determine the impacted elements in these artifacts [24].

- **Dependency Impact Analysis.** The dependency impact analysis focuses on low-level design, compared to the traceability impact analysis. Dependencies in detailed-design and source code are used in order to determine the impacted parts of source code or detailed design. Program slicing [85] and impact analysis on UML models [35] [36] are examples of dependency impact analysis.
- **Experimental Impact Analysis.** Review processes, informal discussions and the application of engineering judgement are defined as experimental impact analysis by Kilpinen [139]. Implicit design dependencies and mechanisms for change propagation can be identified by using expert knowledge in informal discussions. Kilpinen considers experimental impact analysis as unsystematic since there is no tool support and formal methods provided in order to determine the impacted elements.

In the thesis we develop techniques and tools for change impact analysis of requirements and software architecture. Since dependency impact analysis concerns detailed-design and source code, our work is not in the scope of the dependency impact analysis. Our work can be considered within the context of *the traceability impact analysis techniques*.

2.5 Traceability

In this section we analyze the concepts of traceability from various perspectives. We focus on definitions of traceability in requirements engineering and Model Driven Engineering (MDE) (Section 2.5.1). We summarize core concepts of traceability given by von Knethen [142] in order to give the fundamentals of traceability techniques (Section 2.5.2).

2.5.1 Definitions of Traceability

We start with definitions found in literature that considers traceability in general. Next, we give definitions for specific areas.

Traceability is defined in the IEEE Standard Glossary of Software Engineering Terminology [123] 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” and “the degree to which each element in a software development product establishes its reason for existing”. In addition to that the IEEE Standard Glossary [123] simply defines a trace as “a relationship between two or more products of the development process”.

In the domain of requirements engineering, the term traceability is usually used for the ability to follow the traces to and from requirements [262]. One common definition of

requirements traceability is given by Pinheiro [211] as “the ability to define, capture, and follow the traces left by requirements on other elements of the software development environment and the traces left by those elements on requirements”. Similar to this definition, Gotel and Finkelstein [100] define requirements traceability as “the ability to describe and follow the life of a requirement, in both a forwards and backwards direction (i.e., from its origins, through its development and specification, to its subsequent deployment and use, and through periods of on-going refinement and iteration in any of these phases)”. Paige et al. [208] describe traceability as “the ability to chronologically interrelate uniquely identifiable entities in a way that matters. Traceability refers to the capability for tracing artifacts along a set of chained (manual or automated) operations”.

Winkler and Pilgrim [262] discuss the definitions of traceability from perspectives of both requirements engineering and model driven engineering domains in more detail.

We use the definition in the IEEE Standard Glossary of Software Engineering Terminology [123] as our working definition for traceability in the thesis. In this respect our working definition of the term *trace* is that every relation between software development artifacts or between elements in these artifacts can be a trace for a certain traceability purpose like change impact analysis.

2.5.2 Core Concepts of Traceability

In this section we summarize the core concepts of tracing approaches identified by von Knethen et al. [142]. The four core concepts of tracing approaches are the following: (a) purpose, (b) conceptual trace model, (c) process, and (d) tools (see Figure 2.6).

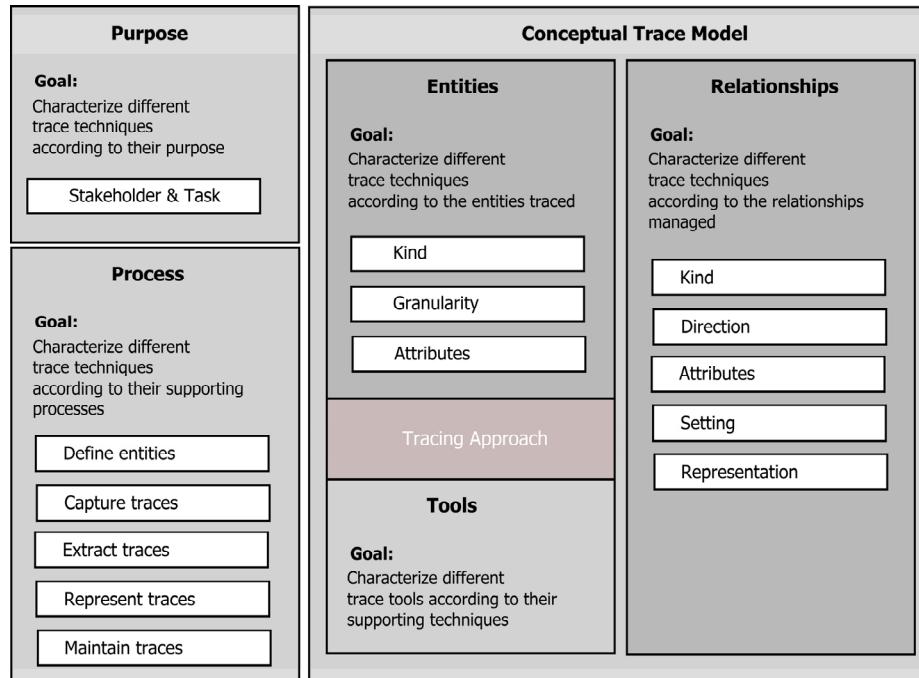


Figure 2.6 Core Concepts of a Tracing Approach [142]

Every trace technique has a purpose such as change impact analysis (see the entity “purpose” in Figure 2.6). The purpose of tracing technique depends on the stakeholder who needs the traceability information and the task of the stakeholder that uses the traceability information.

A conceptual trace model defines what trace entities are and what kind of trace should be captured. The subconcept “entities” characterize different trace techniques according to the kind, granularity and attributes of the entities traced. The characterization of trace techniques in the subconcept “relationship” is based on the kind, direction, attributes, setting and representation of relationships captured as trace. For instance, most of the classification of traceability given in Section 2.5.3 is based on the direction of relationships. The concept “tool support” characterizes trace tools according to what kind of traceability techniques are supported by trace tools.

2.5.3 Classification of Traces

Over the years, various classifications of traces are proposed and emphasized by different sources in literature. The most common ones are *pre-requirements specification*, *post-requirements specification*, *forwards*, *backwards*, *horizontal*, *vertical*, *within-level*, and *between-level* traceability (see Figure 2.7).

Gotel and Finkelstein [100] have introduced *pre-requirements specification (pre-RS) traceability* and *post-requirements specification (post-RS) traceability*. Whereas Pre-RS traceability is concerned with

tracing back from requirements to user needs, Post-RS traceability is concerned with tracing from requirements to design and coding where requirements are realized.

The ANSI/IEEE Std 830–1984 [122] gives the terms *backward traceability* and *forward traceability*. Backward traceability refers to the ability to follow the traceability links from an artifact back to its sources from which it has been derived. Forward traceability describes following the traceability links to the artifacts that have been derived from the artifact under consideration.

Ramesh and Edwards [214] introduce the distinction between *horizontal* and *vertical traceability*. These terms differentiate between traces of artifacts belonging to the same development phase or level of abstraction, and traces between artifacts belonging to different ones.

von Knethen et al. [142] describe a classification which is similar to horizontal-vertical traceability from refinement point of view. They distinguish two types of traces between artifacts on different abstractions as *between-level refinement traces* and *within-level refinement traces*. Within-level refinement traces are between artifact entities at different refinement levels on a certain abstraction level. Such traces are between two system use-cases or between two requirements in the same requirements document. Between-level refinement traces are between entities at different refinement levels on different abstraction levels eg., between a requirement in requirements document and a software component in software architecture design. In the thesis we interpret between-level and within-level refinement traceability as between-model and within-model traces.

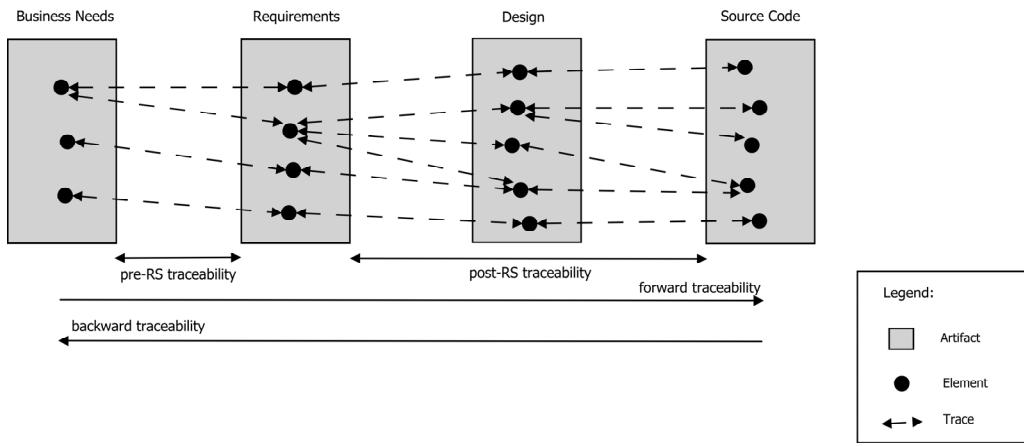


Figure 2.7 Directions of Traces [142]

According to our working definition of trace given in Section 2.5.1, a relation, which is considered as a trace for a traceability purpose, might not be considered as a trace for another traceability purpose. In the thesis we do not use any classifications given above since

a relation which is considered a trace in above classifications might not be a trace for change impact analysis in the thesis.

2.6 Model Driven Engineering

The concept of Model Driven Engineering (MDE) was introduced as a generalization of the Model Driven Architecture (MDA) for software development. In this section we first introduce MDA and then explain MDE.

2.6.1 Model Driven Architecture

MDA is a software development approach proposed by Object Management Group (OMG). The MDA Guide [199] provides definitions of concepts used in MDA.

MDA aims at solving the problem of continuous change of software technologies that forces software development companies to port their solutions every time a new technology appears. For instance, Java platform [129] was announced as an object-oriented software development environment in 1990s and many software development companies developed their solutions in this platform. .Net platform [190] was proposed as a competitor of Java platform in 2000s. As a result of market trends, some software companies switched to .Net platform and had to port their implemented solutions for .Net platform. Such kind of changes in technologies creates a problem with *portability* which may require significant efforts. MDA proposes the use of models of the same system at different abstraction levels with conversions between the models to solve the *portability* problem.

To cope with the portability problem MDA uses a set of concepts such as *model*, *metamodel* and *transformation* with a classification of models as Computation Independent Models (CIMs), Platform Independent Models (PIMs) and Platform Specific Models (PSMs). A CIM is a view of a system from the computation independent viewpoint. The PIM focuses on the operation of a system while it still hides the details necessary for the implementation of the system in a particular platform. The PIM specifies a degree of platform independency to be suitable for use with a number of different platforms of similar type. On the other hand the PSM includes details of the platform implementation.

The MDA Guide [199] defines model as “a model of a system is a description or specification of that system and its environment for some certain purpose. A model is often presented as a combination of drawing and text. The text may be in a modeling language or in a natural language”. The MDA Guide defines metamodel as “model of models”. A more detailed definition of metamodel by FRISCO report is that “metamodel is a model of the conceptual foundation of a language, consisting of a set of basic concepts, and a set of rules

determining the set of possible models denotable in that language” [76]. Seidewitz [230] defines a metamodel as “a model of models expressed in a given modeling language”. We use the definition by Seidewitz as our working definition for metamodel in the thesis.

Models are organized in a hierarchy that spans multiple levels. The organization of levels is referred to as meta-modeling architecture. Figure 2.8 gives an example of meta-modeling architecture with three levels. At the bottom level there are models expressed in various modeling languages. This level is called *model level*. An example model in this level is *Model_L* expressed in a modeling language called *L*. Metamodels of the languages form the second level in the stack called *metamodel level*. The metamodel of *L*, *LModel_{ML}*, is expressed in another language called *Metalanguage (ML)*. The metamodels of the languages that express metamodels form the third level called *metametamodel level*. There is *InstanceOf* relation between a metamodel of a language and models expressed in that language. The levels can be formed infinitely with *InstanceOf* relation. However, in practice only three levels are used. The top level contains a self-reflective model. The model *MLModel_{ML}* in Figure 2.8 is expressed in the *ML* language itself (the self *InstanceOf* relation).

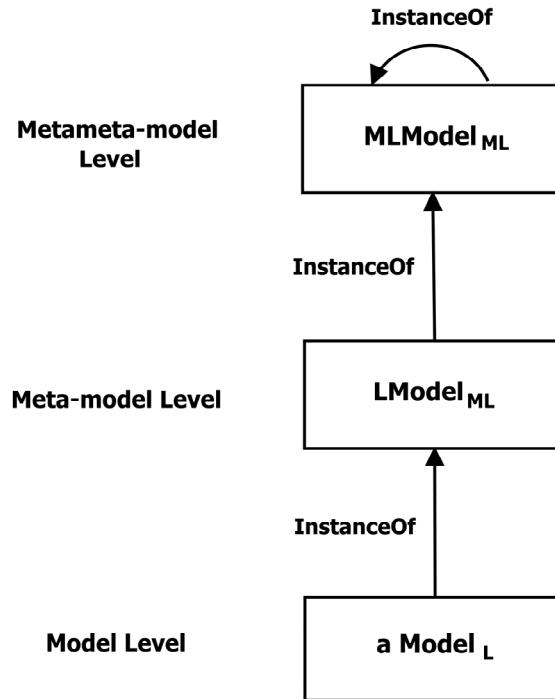


Figure 2.8 Meta-modeling Architecture

The basic operation applied on models in MDA is *model transformation*. The MDA guide defines model transformation as “the process of converting one model to another model of the same system”. The transformation pattern between models is given in Figure 2.9.

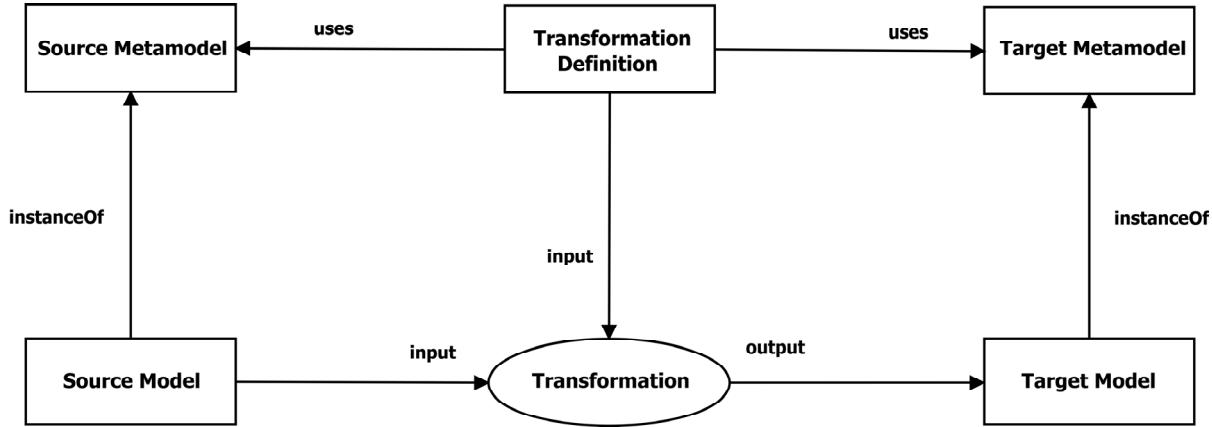


Figure 2.9 Transformation Pattern

A transformation definition is capable of transforming a set of source models. A transformation that transforms source models expressed in a source metamodel to models expressed in a target metamodel uses the meta-entities defined in the source and target metamodels.

2.6.2 Model Driven Engineering

Model Driven Engineering (MDE) is a generalization of MDA by adding the notion of software development process to MDA.

MDA considers the classification of models based on only the level of model abstraction. CIMs and PIMs can be considered at a higher abstraction level than PSMs. MDE utilizes the use of Domain Specific Modeling Languages (DSML) on the base of different distinctions of models such as the subject area models belong to or organizational issues. The number of distinctions of models is not limited and depends on the needs in a software development project. MDE technologies combine the following [227]:

- **Domain Specific Modeling Languages.** They are used to model the application structure, behavior and requirements within particular domains such as financial services, embedded systems [227]. Similar to MDA, metamodels are used to describe DSMLs by defining the entities for the concepts and relationships between these concepts in the domain with clear semantics and constraints.
- **Transformation Engines and Generators.** Definition of metamodels is required but not sufficient for a complete MDE. We have to define transformations between metamodels of DSMLs to obtain the main artifacts of MDE: target models. In addition to that transformation engines and generators are used to analyze certain

aspects of models and synthesize various artifacts like design models and source codes [227].

With these two techniques above, MDE aims at detecting and preventing errors early in the software development life cycle by using domain specific constraints and performing model checking.

2.7 Survey of Traceability in MDE

In this section, we discuss the state-of-the-art in traceability approaches in MDE and appraise them with respect to four general comparison criteria: *representation*, *mapping*, *change impact analysis* and *tool support*. These comparison criteria are influenced by the core concepts of tracing approaches (*purpose*, *conceptual trace model*, *process*, and *tools*) provided by van Knethen [142]. Change impact analysis is our tracing purpose in the thesis. Mapping and representation are considered as a part of the conceptual trace model to characterize trace techniques for entities and relations to be traced.

2.7.1 Traceability Approaches in MDE

The traceability approaches we analyze are classified into three categories: *requirements-driven approaches*, *modeling approaches* and *transformation approaches*. The requirements-driven approaches consider requirements of the system as a starting point for traceability. The modeling approaches investigate how metamodels and models are involved in tracing processes. Transformation approaches make use of model transformation mechanisms for generating trace information.

2.7.1.1 Requirements-Driven Approaches

In the field of Requirements Engineering, Gotel and Finkelstein [100] define traceability as the ability to describe and follow the life of a requirement, in both forward and backward specification, to its subsequent deployment and use, and through periods of ongoing refinement and iteration in any of these phases. Tracing requirements in both forward and backward directions helps stakeholders and developers to understand requirements in more detail. The following subsections present five requirements-driven approaches.

2.7.1.1.1 Requirements Traceability and Transformation Conformance (RTTC)

Almeida et al. [11] aim at simplifying the management of traces between requirements and various design artifacts. They propose a framework as a basis for tracing requirements, assessing the quality of model transformation specifications, metamodels and models. The

framework allows designers to relate requirements in the early stage of the development to the various products of the model-driven design process.

Traceability cross-tables are used for representing relationships between application requirements and models, considering different model granularities. Since model-driven techniques consist of different abstraction levels like platform-independent and platform specific levels, Almeida et al. propose a notion of conformance between models to trace requirements throughout abstraction levels. Change impact analysis in requirements is deferred to future work.

2.7.1.1.2 Event Based Traceability (EBT)

Event-Based Traceability (EBT) [50] is a method for automating trace generation and maintenance. In EBT, requirements and other traceable artifacts, such as design models, are no longer directly related, but linked through publish-subscribe relationship based on *Observer design pattern* [88].

The main components of the system are the *event server*, *requirements manager* and *subscriber manager*. The requirements manager is responsible for triggering change events by publishing an event message when a change occurs. Event messages carry structural and semantic information concerning the change context. The event server is primarily responsible for managing subscriptions, receiving event messages from the requirements manager, and forwarding customized event messages to the subscriber manager. The subscriber manager resolves event notifications and restores related artifacts and traces to a new state if necessary.

2.7.1.1.3 Goal Centric Traceability (GCT)

Cleland-Huang et al. [52] introduce a goal-centric approach for managing impact of a change in non-functional requirements. Goal Centric Traceability (GCT) models non-functional requirements and their dependencies using a Softgoal Interdependency Graph (SIG).

The approach has four steps to analyze and implement changes on dependent artifacts: *goal modeling*, *impact detection*, *goal analysis*, and *decision making*. In goal modeling, goals are decomposed into subgoals to reflect the fact that dependencies exist between various non-functional requirements (represented by softgoals). To understand the trade-offs among non-functional requirements, the subgoals are decomposed into operationalizations providing candidate solutions for the goal. In the impact detection, when a change occurs in non-functional requirements, a probabilistic retrieval algorithm dynamically returns related traces in the SIG. In the goal analysis the user modifies the contributions, from the impacted goal elements to their parents. For each impacted element, changes are propagated

throughout the SIG to identify potentially impacted goals. In the decision making it is determined if the change should be implemented or not. Stakeholders evaluate the impact of the proposed change in non-functional requirement goals.

2.7.1.1.4 Event Based Traceability with Design patterns (EBT-DP)

In [51], Cleland-Huang and Schmelzer introduce another requirements-driven traceability approach. Their work is based on EBT [50] but they describe a different process for dynamically tracing non-functional requirements to design patterns. The process is divided into two phases.

During the initial phase, user-defined traces are established. Design elements are traced to a cluster, which is the application of the design pattern. Then, a trace is established between the non-functional requirement and the cluster. Therefore, the number of traces between design artifacts and non-functional requirements is decreased. In the second phase, the well established descriptions and invariant rules of a design pattern permit the automatic and dynamic generation of code (from the pattern to specific class implementations). By establishing traces between requirements and the cluster, the approach aims at minimizing the cost and effort of establishing and maintaining traceability links.

2.7.1.1.5 Reference Models for Requirements Traceability (RMRT)

Ramesh and Jarke [215] provide an empirical approach and focus on interviews conducted in software organizations to study a wide range of traceability practices. As a result of the study, Ramesh and Jarke constitute reference models that include the most important kinds of traces for various software development artifacts.

One of the main motivations behind the study is to capture traceability needs of stakeholders and present reference models for each need. Ramesh and Jarke classify the participants of the study as high-end and low-end users of traceability practices. Trace models are presented to reflect the trace entities captured by high-end and low-end users, and then a set of five reference models is customized. Requirements are considered as traceable entities in all these reference models.

2.7.1.2 Modeling Approaches

In MDE, trace metamodels are crucial to store and represent traces, derived from dependencies between source and target elements. Modeling approaches represent trace information as models. As an instance, the UML profile mechanism gives a solution to store and represent traces. There is also a standard stereotype for traceability in UML [201].

2.7.1.2.1 Scenario Driven Approach to Trace Dependency Analysis (SDTDA)

Egyed [70] presents an automated approach for generating and validating traces. He addresses the problem that the absence of trace information or the uncertainty of trace correctness limits the usefulness of software models. The proposed approach reduces the complexity of trace generation and validation by using test scenarios and hypothesized traces. The approach requires an observable and executable software system, design artifacts, scenarios describing test cases, and a set of initial hypothesized traces linking development artifacts and scenarios.

Executing test scenarios in the running system leads to traces between scenarios and source code. The runtime behavior of the scenarios is translated into a footprint graph. Traces are generated and validated by using the rules that characterize how the footprint graph relates to the hypothesized traces and artifacts to which they are linked.

2.7.1.2.2 Operational Semantics for Traceability (OST)

Aizenbud-Reshef et al. [6] present an approach which defines an operational semantics for traceability in UML. Three main issues for traceability are stated: querying (e.g. impact analysis, coverage queries), following traces along the life-cycle of a project, and keeping the system and its documentation up to date. Two types of semantics based on these issues are presented: *preventative semantics* and *reactive semantics*. Preventative semantics describes things that should not happen; reactive semantics describes what should be triggered when something happens to one or more of the related elements or to the relationship itself.

Operational semantics of a trace is defined by a set of semantic properties. A semantic property is a triplet (*event*, *condition*, and *actions*). Event involves an element of the trace. Condition is a logical constraint and actions can be either preventative or reactive actions.

2.7.1.2.3 Unifying Traceability Specification Scheme (UTSS)

Limon and Garbajosa [156] analyze current traceability schemes in order to obtain relevant features and identify overlaps and inconsistencies among the approaches. Based on the analysis, they propose a traceability scheme specification approach to facilitate traceability specification for a given project, to improve the traceability management, and to automate some trace management processes.

2.7.1.2.4 Precise Transformation Traceability Metadata (PTTM)

Vanhooff and Berbers [252] provide a UML profile for transformation traceability metadata in order to reason about past transformations. Transformation traceability links provide a complete or partial history of model changes caused by the transformations. Transformation

traceability metadata are used to make individual transformation units more modular and easier to maintain.

Vanhoooff and Berbers list four important requirements for their approach. At first, the transformation traceability information should be kept by all transformation units. Secondly, traces should be extended with transformation unit specific information. Another requirement is that all information should be kept in a UML model itself and, at last, it should be possible to easily add traces manually for non-automatic transformations.

2.7.1.3 Transformation Approaches

Model transformations are considered as a mechanism which supports automating both generation and validation of traces between models. Hence, most of the transformation languages support automatic generation of traces.

2.7.1.3.1 Loosely Coupled Traceability (LCT)

Jouault [134] shows how traceability can be added to transformation programs written in the ATLAS Transformation Language (ATL) [136] in order to overcome the limits of implicit traceability. The trace generation mechanism of ATL is implicit. Such a form of traceability does not persist after executing a transformation.

Jouault considers the traceability information as an additional target model of a transformation program. His approach supports generating traces in the same way other target model elements are generated. Jouault provides a higher-order transformation (HOT) that transforms ATL transformations to insert the trace creation code to the transformations. One of the advantages of the solution is that trace generation code is not tightly coupled to transformation logic.

2.7.1.3.2 On Demand Merging of Traceability (ODMT)

Kolovos et al. [145] present an approach for merging trace models with other software development models. The correspondences between elements of the source models are established and then corresponded elements are merged. The Epsilon Merging Language (EML) [144] is used to implement model merging with traces. EML is a plug-in for the Eclipse and supports managing EMF and MOF models as well as XML documents.

2.7.1.3.3 Traceability Framework for Model Transformations (TFMT)

Falleri et al. [77] proposes a traceability framework, implemented in Kermeta [176]. The framework allows tracing transformation chains within Kermeta, by means of the specification and implementation of a language independent trace metamodel. Falleri et al. have implemented the following features of the traceability framework [77]: generic

traceability items, trace serialization, and a simple transformation for trace visualization using Graphviz [101] (in order to visualize the resulted transformation trace chain).

2.7.2 Evaluation of the Approaches

In this section we present a comparative analysis of traceability approaches for MDE with respect to the following comparison criteria: *representation of traceability information, mapping models, change impact analysis, and tool support*.

2.7.2.1 Representation

The capability of the approaches to represent traces is evaluated in Table 2.1.

Table 2.1 Representation of Trace Information in Traceability Approaches in MDE

		Representation
Requirements-Driven Approaches	RTTC	Traceability cross-table
	EBT	Event-based subscriptions
	GCT	Softgoal Interdependency Graph (goals, operationalizations and contribution links) and traceability matrix
	EBT-DP	Softgoal Interdependency Graph and event-based subscriptions
	RMRT	Traceability metamodels
Modeling Approaches	SDTDA	Footprint graphs
	OST	Rules, conditions and actions
	UTSS	Traceability Scheme (TS)
	PTTM	UML models
Transformation Approaches	LCT	Trace model
	ODMT	EML (the metamodel) and UML (the trace model)
	TFMT	Kermeta models (the proposed metamodel) and XMI (the serialized instances of transformation chain)

RTTC: Almeida et al. [11] represent traceability information for application requirements by using cross-tables. Assesment activities or conformant transformations between models are necessary to justify check marks in cross-tables.

EBT and EBT-DP: Event-based subscriptions are used to represent traces in EBT [50] and EBTDB [51]. The notification of the events carries structural and semantic information concerning a change context. As EBT-DB [51] considers SIG models, traces are also represented by interdependencies between softgoals (non-functional requirements) and operationalizations (representing design patterns).

GCT: GCT [52] uses softgoal interdependency graphs in order to trace between goals and their operationalizations. A traceability matrix is also constructed to relate SIG elements with classes.

RMRT: In RMRT [215], traceability reference models are used to represent traces. Granularity of traces depends on the expectations of the stakeholders. RMRT represents simple or more detailed traces across the low-use and high-use reference models. Implementations of the reference models present distinct ways to embody traceability information.

SDTDA: In [70] , traces are represented in traceability matrix and a graph structure called footprint graph. The runtime behavior of test scenarios is translated into a footprint graph. The footprintgraph is interpreted via a set of rules in order to generated new trace information. Final representation of generated traces is done in traceability matrix.

OST: In [6], semantic properties (events, conditions and actions) are used to capture and represent traces.

UTSS: In [156], Limon and Garbajosa analyze several traceability approaches and propose a unified Traceability Scheme (TS) specification. TS is composed of a dataset, a type set, a minimal set of traces, and a metrics set for the minimal set of traces.

PTTM: Vanhooff and Berbers [252] provides a UML profile to represent traces. Using stereotypes and tagged values, they add trace semantics to existing UML elements.

LCT: Jouault [134] considers traceability information as a model and extends ATL programs to provide trace generation during model transformations. Traces generated by model transformations are represented as Ecore models.

ODMT: Kolovos et al. [144] use an EML trace metamodel for merging, which is compliant with Meta-Object Facility (MOF). UML diagrams are used as example models in the approach.

TFMT: In TFMT [77], traces are represented as Kermeta models and instances of resulting transformation trace chains are serialized as XMI.

2.7.2.2 Mapping

The mapping criterion analyzes whether the approach is capable of supporting traces among models at different levels of abstraction. The traceability approaches are evaluated for mapping, based on *intra-level relationships* (traces among artifacts of the same abstraction level), *inter-level relationships* (traces among artifacts of different abstraction levels), or both *intra and inter-level relationships* (see Table 2.2).

Table 2.2 Mapping, Change Impact Analysis and Tool Support in Traceability Approaches

		Mapping	Change Impact Analysis	Tool Support
Requirements-Driven Approaches	RTTC	inter	no	no
	EBT	inter	yes	yes
	GCT	intra & inter	yes	partially
	EBT-DP	intra & inter	yes	yes
	RMRT	intra & inter	yes	yes
Modeling Approaches	SDTDA	intra & inter	no	partially
	OST	inter	no	no
	UTSS	intra & inter	no	no
	PTTM	intra & inter	no	no
Transformation Approaches	LCT	intra & inter	no	yes
	ODMT	inter	no	yes
	TFMT	intra & inter	no	yes

RTTC: RTTC [11] supports traces from requirements models to other models at different levels of abstraction.

EBT and EBT-DP: Both EBT [50] and EBT-DB [51] support mapping requirements to other artifacts, by using event-based mechanism.

GCT: GCT [52] provides traces between softgoals and operationalizations at the requirements level, by using the softgoal interdependency graph. Requirements are traced to source code by using traceability matrix.

RMRT: In RMRT [215], intra-level and inter-level traceability are supported by the low and high-use metamodels, which provide mappings between requirements and many other elements (system objectives, system components, functions, etc).

SDTDA: The trace types in [70] provide both intra-level and inter-level mapping. SDTDA supports both forward and reverse engineering.

OST: The approach in [6] is proposed for UML models but there is no indication about supporting traces for UML models at different abstraction levels.

UTSS: In [156], it is stated that the minimal set of traces of the unified traceability schema must consider traces among artifacts themselves, as well as traces among a set of artifacts and the artifacts of a previous (or next) development phase.

PTTM: In PTTM [252], a transformation traceability metamodel is mapped to UML profiles. UML models at different abstraction levels can be traced.

LCT: The trace metamodel presented in [134] allows establishing traces between models at the same abstraction level or different abstraction levels.

ODMT: The approach in [144] only presents a traceability method for unidirectional and inter-level traces.

TFMT: The approach in [77] supports forward, backward, intra-level and inter-level traceability, depending on the definition of source and target models in the transformation.

2.7.2.3 Change Impact Analysis

The change impact analysis criterion checks whether an approach determines the effect of change on the entire system and on the artifacts across the software development lifecycle. Table 2.2 shows evaluation of the approaches for change impact analysis.

EBT: A set of standard change events is defined for recognition and publication of change events [50]. A method for monitoring user's actions is proposed.

GCT: The GCT [52] provides change impact analysis among functional and non-functional requirements, represented by using softgoal interdependency graphs.

EBT-DB: EBT-DB [51] supports the identification of critical elements that should remain in the system for keeping the integrity of a traceable non-functional requirement.

RMRT: Ramesh and Jarke [215] provide change impact analysis based on the description of the rationale submodel.

Other approaches [6] [11] [70] [77] [134] [144] [156] [252] do not support change impact analysis.

2.7.2.4 Tool Support

Tool support is fundamental for application of a traceability method, not only for visualization and management of manually or automatically traces, but also for proper reasoning support on trace information. Table 2.2 summarizes tool support of the approaches.

EBT: EBT [50] has a client-server architecture using Observer design pattern. The event trigger mechanism is implemented on top of DOORS [120] to capture change events.

GCT: The GCT model [52] has partial tool support. Despite of the fact that the retrieval algorithm uses probability to return traces, user's appraisal is required to manage traces.

EBT-DP: A few features of EBT-DP [51] (generation of traces) are implemented.

RMRT: The reference metamodels for traceability by Ramesh and Jarke [215] are encoded in a knowledge-based meta database management system called ConceptBase. The metamodels are also adopted in several commercial tools, such as SLATE [240].

SDTDA: The activities for scenario-testing and finding hypothesized traces in [70] are manual; trace analysis and result interpretation are automated.

LCT: LCT [134] is implemented in ATL [134] [136].

ODMT: ODMT [144] is implemented in EML. EML [145] is used to implement merging models with trace models.

TFMT: The transformation chain trace metamodel is supported by Kermeta [176] and graphical visualization of traces is provided in Graphviz [101].

The other evaluated approaches [6] [11] [156] [252] do not provide any tool support.

2.7.3 Open Issues for Traceability in MDE

From the comparative analysis of the approaches we identify the following open issues:

- ***Open Issue 1: Automation in the early development stages.*** In the early development stages like requirements analysis and architectural design, less automation is provided to cope with traceability.
- ***Open Issue 2: Trace semantics.*** Most of the approaches that we surveyed do not focus on the use of trace semantics. How can trace semantics be formalized and represented? How can trace semantics be used to achieve traceability goals such as change impact analysis?
- ***Open Issue 3: Incremental model transformation.*** Incremental model transformation [65] [110] [132] [146] is an active research topic in MDE. The use of traces with incremental model transformations is partially known. There are still some questions not answered. For instance, how can traces between source and target models be used to determine parts of incremental transformation to be re-executed in case of a change?
- ***Open Issue 4: Trace generation from implicit trace information.*** Trace information may not always be encoded in a dedicated structure (implicit trace information). Most of the approaches do not explore mechanisms for generating traces from implicit trace information.
- ***Open Issue 5: Scalability of traceability tools.*** Since software projects become larger during their development, and the software specification contains heterogeneous artefacts, scalability is an important criterion to be considered when evaluating the use of traceability approaches. However, most of the traceability tools are research prototypes and scalability of these tools is not explored.
- ***Open Issue 6: Maintenance of traces.*** Most of the approaches investigate the use of traceability to determine the model elements impacted by a change. However, maintenance of existing traces after changes to artifacts is still an open issue for most of the approaches. Trace maintenance is very important to ensure correctness of traces.

Some of the open issues (*Open Issues 1, 2, 4 and 6*) are detailed further in Chapter 3; some open issues (*Open Issue 3* and *Open Issue 5*) are not addressed in the thesis at all.

2.8 Conclusions

In this chapter we introduced the basic concepts in requirements engineering, software architecture design and analysis, software change management, traceability and MDE. Our approach for change management for requirements and software architecture is based on definitions found in literature and selection of those definitions that suit the objectives of the thesis.

This chapter also presented a survey of traceability techniques in MDE in which we identified some open issues. Some of these open issues are addressed in the rest of the thesis. One of the open issues addressed is trace semantics. We explore the possible applications of trace semantics in change impact analysis for requirements and software architecture.

In this chapter we answered Research Question 1 (*What does traceability mean? Can every relation between software development artifacts or between elements in the artifacts be a trace? What is the criterion for a relation to be a trace?*) and Research Question 2 (*What are the current traceability approaches for change management? What are their deficiencies? Which solutions and technologies have been proposed to address these deficiencies?*) raised in Chapter 1. Our working definition of the term *trace* is that every relation between software development artifacts or between elements in these artifacts can be a trace for a certain traceability purpose like change impact analysis. In Section 2.7, we presented current traceability approaches with a comparative analysis. The open issues for traceability in MDE in Section 2.7 are based on the deficiencies of the current traceability approaches.

Chapter 3

3 Analysis of Impacts Explosion in Traceability

In this chapter, we motivate the need for semantics of traces between requirements, and requirements & architecture for change management by exploring impacts explosion problem with some change scenarios.

3.1 Introduction

This chapter gives a detailed analysis of the impacts explosion problem that we address in the thesis. Our traceability goal is change impact analysis for requirements and software architecture, for example, determining which requirements and architectural elements are impacted by a change of requirements. Impacts explosion problem is originally formulated in the general case for software life-cycle objects by Bohner [22] [23] [24] [25].

In this chapter we answer *Research Question 3* raised in Chapter 1: *What are the change scenarios for requirements and software architecture? What is necessary for these change scenarios to be handled? Which solutions can be used?* We first explain the impacts explosion problem for requirements and software architecture. We identify some change scenarios where we may have change impacts explosion.

The structure of the chapter is as follows. Section 3.2 describes impacts explosion problem, in general, as formulated by Bohner. In Section 3.3 we illustrate specifics of the impacts explosion problem for requirements and software architecture. Section 3.4 discusses the change scenarios. In Section 3.5, the summary of the problems is given. Section 3.6 concludes the chapter.

3.2 Impacts Explosion Problem

Change impact analysis is defined by Bohner [22] [23] [24] [25] as “identifying the potential consequences of a change, or estimating what needs to be modified to accomplish a change”. A change in a software system may affect other parts of the system and the change may trigger ripple-effects which cause direct and indirect impacts on other elements [24]. The relationships between elements are considered as traces. A direct impact occurs when the affected element in the artifact is directly linked with one trace to the changed element. An indirect impact occurs when the affected element in the artifact is indirectly linked with more than one trace to the changed element. Figure 3.1 shows an example directed graph of software life-cycle objects (SLO) with traces. Software life-cycle objects stand for elements in development artifacts (e.g. requirements in requirements documents, classes & methods in code, components in architecture).

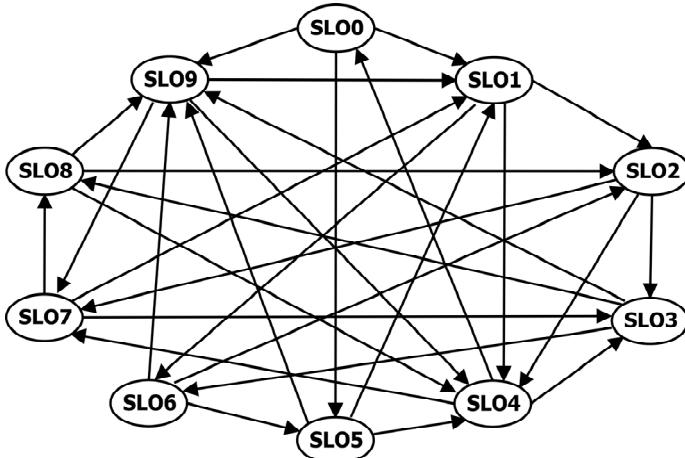


Figure 3.1 Simple Directed Graph of Software Life-Cycle Objects [23]

When a change occurs in SLO1 in Figure 3.1, SLO2 has a direct impact and SLO3 has an indirect impact by the change. Table 3.1 gives the traces between software life-cycle objects in Figure 3.1 in a connectivity matrix.

Table 3.1 Connectivity Matrix of Traces [23]

	SLO0	SLO1	SLO2	SLO3	SLO4	SLO5	SLO6	SLO7	SLO8	SLO9
SLO0		x				x				x
SLO1			x		x		x			

SLO2			x	x			x	
SLO3						x		x
SLO4	x		x				x	
SLO5		x			x			x
SLO6			x			x		x
SLO7		x		x				x
SLO8			x		x			x
SLO9		x			x		x	

Connectivity matrix of traces is transformed into a reachability matrix where the objects, which can potentially be affected by a change to a particular SLO, are indicated [23] (see Table 3.2). Reachability matrix denotes traces inferred by using transitive closure of traces in the connectivity matrix.

Table 3.2 Reachability Matrix of Traces [23]

The reachability matrix indicates both direct and indirect impacts on software life-cycle objects. For instance, the direct impact in SLO2 and the indirect impact in SLO3 of the change in SLO1 in Figure 3.1 can be inferred in the reachability matrix in Table 3.2. However, transforming the connectivity matrix into a reachability matrix does not gain any additional information since every object is related directly or indirectly to every other object in the matrix. Bohner suggests the use of the notion of distance between SLOs in order to limit the detection of impacts (see Figure 3.2). The notion of distance in Figure 3.2 explains how the number of impacts explodes.

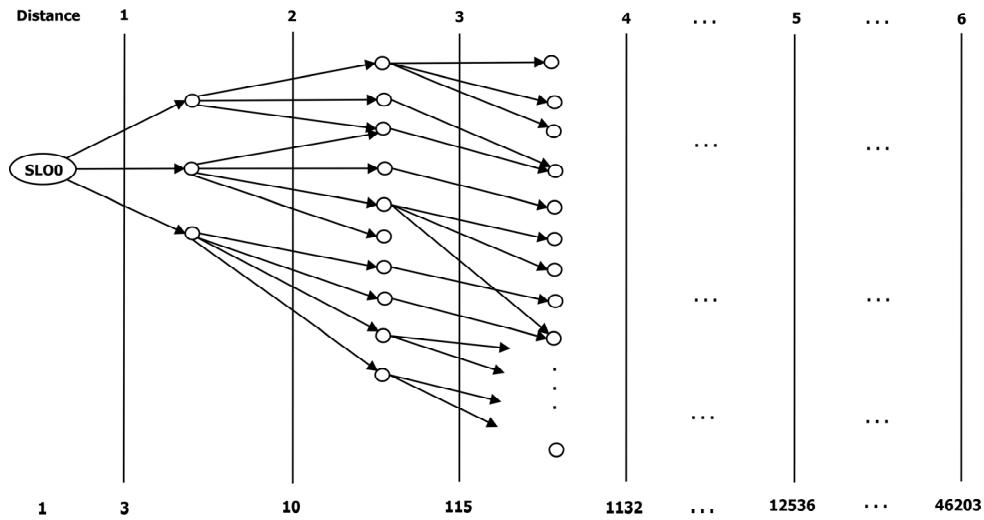


Figure 3.2 Impacts Explosion without Semantics [25]

After a change is introduced to SLO0, 3 impacts are introduced at a distance of 1. The number of impacts jumps to 10, 115, 1132 and 46203 at the distances of 2, 3, 4, 5 and 6 with only approximately nine traces per SLO [25]. Bohner [24] states that change impact analysis must employ additional semantic information to increase the accuracy by finding more valid impacts and reducing the number of false-positive impacts. The use of trace semantics in impact analysis can identify some of the unimpacted software life-cycle objects at the initial distances and this prevents impact explosion at the later distances.

3.3 Impacts Explosion in Requirements and Software Architecture

Requirements and architectural elements are considered as Software Life-cycle Objects (SLO). In current practice, requirements are textual artifacts with structure often not explicitly specified. Relations between requirements are mostly not documented. Table 3.3 represents a part of a requirements document for a Course Management System (CMS). The

requirements are about the CMS for a school which has features such as notification of students about exam grades and messaging for communication at school.

Table 3.3 Some Requirements for a Course Management System

R1: The system shall notify students about exam grades.
R2: The system shall provide e-mail messaging.
R3: The system shall provide sms messaging.
R4: The system shall provide sms and e-mail messaging.
R5: The system shall allow lecturers to create courses.
R6: The system shall allow lecturers to specify enrolment policies based on grade.
R7: The system shall allow lecturers to manage course information.
R8: The system shall allow lecturers to specify enrolment policies based on grade.

There are implicit relations between requirements in Table 3.3. For instance, the system needs messaging in order to notify students about exam grades. The system property given in R1 *requires* the system property given in R4.

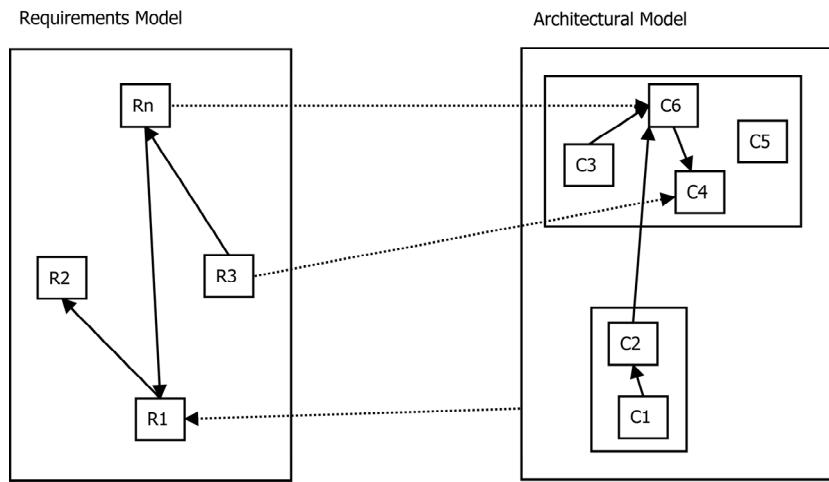


Figure 3.3 Requirements and Architectural Models with Traces

We need explicit structure of requirements and requirements relations in order to do change impact analysis. Requirements metamodels and models can be used to provide an explicit structure to requirements documents with relations between requirements. Requirements in

a requirements model can be linked to architectural elements in an architectural model with traces. Figure 3.3 shows requirements and architectural models with traces.

Any relation between requirements, architectural elements, and requirements & architectural elements plays a role of trace in change impact analysis. For example in Figure 3.3, a change in requirement R_3 has a direct impact on architectural component C_4 , and an indirect impact on component C_6 through the relation of R_3 and R_n . The general impacts explosion problem described for the software life-cycle objects by Bohner [24] is valid for elements in requirements and architectural models. There might be multiple reasons of impacts explosion in requirements and software architecture:

- Impacts explosion might happen due to large highly connected systems having bad decomposition. The requirements and architecture of the system might be decomposed in such a way that every element in the requirements and architectural models is connected. Therefore, changing a requirement might affect every element in the models.
- Impacts explosion might happen due to the lack of semantic information. Every requirement and architectural element directly/indirectly related to the changed requirement might be identified as a candidate impacted element due to the lack of trace semantics. Some of the candidate impacts might be false positives which cause the impacts explosion. In the thesis, we address the impacts explosion in requirements and software architecture due to the lack of semantic information.

When a change is introduced to a requirement, we first want to determine if there is any other impacted requirement. After we find out all impacted requirements, we need to identify the architectural elements impacted by the change in the requirement.

In order to determine the impacted requirements in the requirements model, we can form a connectivity matrix and then a reachability matrix for requirements relations. The reachability matrix for requirements relations will be mostly the same with the one in Table 3.2 which indicates every SLO might be impacted.

In a reachability matrix like the one in Table 3.2, the requirements engineer may have to analyze all requirements in the model for a single change. This may result in neglecting the actual impact of a change in the requirements model. Manual inspection is error-prone and time consuming. Consequently, the cost of implementing a change in the requirements model may become several times higher than expected.

After identifying impacted requirements in the requirements model, impact analysis is applied to the architectural model in order to determine architectural elements impacted by the requirements change. Then, we form a connectivity matrix and a reachability matrix for traces between requirements and software architecture. Again, like the one in Table 3.2 there is a high possibility of having a reachability matrix which indicates that every architectural element in the software architecture might be impacted. The consequences of change impact analysis indicated for requirements models are also valid for change impact analysis in software architecture.

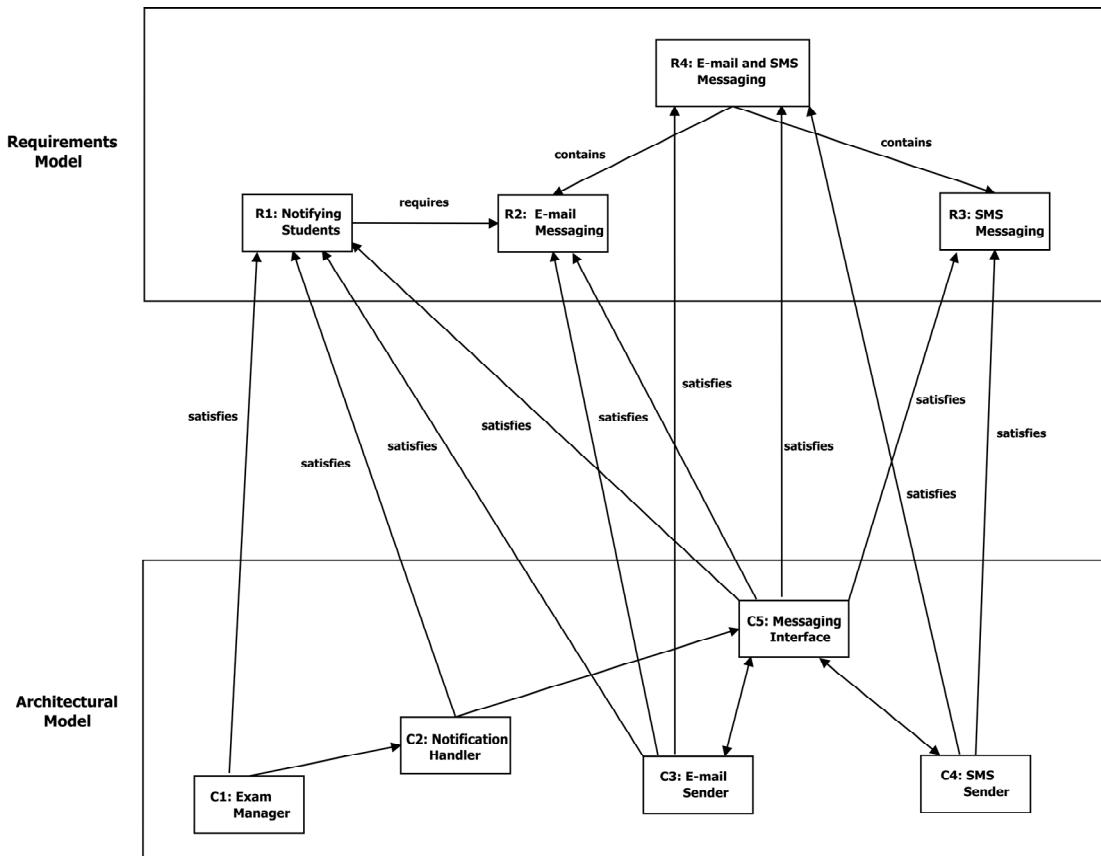


Figure 3.4 Part of Requirements and Architectural Models for Course Management System

In Figure 3.4, we give example requirements and architectural models of the CMS. Four requirements (notifying students, e-mail messaging, sms messaging, and e-mail & sms messaging) in the requirements document in Table 3.3 are given with their relations in the requirements model. Assume that there is a change request for the CMS. Audio messaging is requested for communication at school. The requirement R4 (E-mail and Messaging System) in Figure 3.4 is updated for the change request. By following direct and indirect requirements relations without using any semantic information, it is found that all requirements (R1, R2,

R_3 and R_4) and architectural elements (C_1, C_2, C_3, C_4 and C_5) satisfying these requirements are candidate impacted.

By knowing the semantics of the requirements change and traces, we can eliminate some of the false positive candidate impacted requirements and architectural elements. For instance, the change for the requirements in Figure 3.4 is adding a new property (an audio messaging) to R_4 which does not have any impact on existing system properties. R_1, R_2 and R_3 are not impacted by the change in R_4 . Only the architectural elements (C_3, C_4 and C_5) are candidate impacted. Without making the semantics of changes and traces explicit in the models, the requirements engineer and software architect have to make these reasoning by themselves. The traceability tools and techniques employing additional semantic information can assist the requirements engineer and software architect to do impact analysis on models.

3.4 Change Scenarios for Change Impact Analysis

In this section we explain when we have the impacts explosion problem in change management for requirements and software architecture. We identified two change scenarios in which requirements evolve and then software architecture is updated for the changes in requirements. Figure 3.5 shows the requirements and architectural models with traces for requirements evolution.

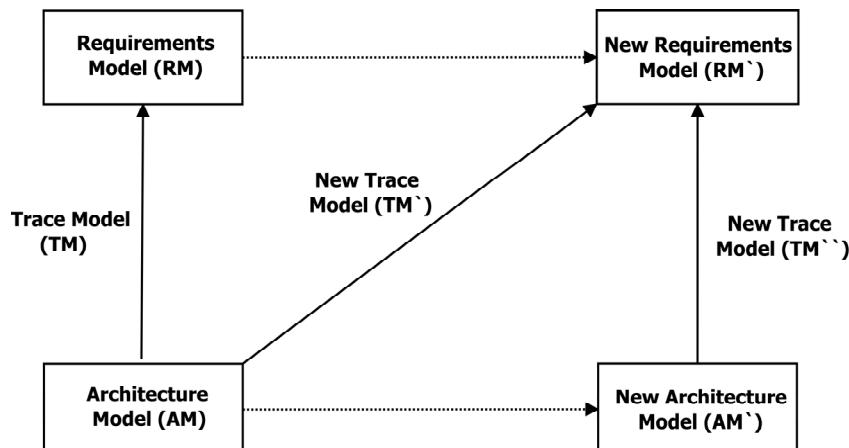


Figure 3.5 Requirements and Architectural Models with Traces for Requirements Evolution

Section 3.4.1 and Section 3.4.2 explain the change scenarios by using requirements and architectural models with traces in Figure 3.5.

3.4.1 Scenario 1: Requirements Evolve

The steps in Scenario 1 are the following:

- *Receiving the change request:* The change request for requirements in the Requirements Model RM in Figure 3.5 is received by the requirements engineer.
- *Performing change impact analysis in the Requirements Model RM:* The requirements engineer interprets the change request as a set of changes in the RM. The impact of each change is analyzed by propagating the change in the RM.

During the change propagation, the requirements engineer may have to encounter explosion of impacts where he has to investigate all requirements in the model RM.

- *Updating the Requirements Model RM:* After the impact analysis, it is decided which changes will be applied to the RM. If any, then the RM is updated (New Requirements Model – RM').
- *Verifying the Architectural Model AM:* After updating the RM, the software architecture (Architectural Model - AM) is verified in order to determine if it satisfies the new/changed requirements
- *Updating the Architectural Model - AM:* If there is any requirement not satisfied by the Architectural Model AM, the AM is updated to make the new/changed requirements being satisfied (New Architectural Model – AM').

The software architect might analyze the impact of the requirements change in the Architectural Model AM to update the AM where he might encounter impacts explosion.

3.4.2 Scenario 2: Requirements and Software Architecture Evolve

The steps in Scenario 2 are the following:

- *Receiving the change request:* The change request for requirements in the Requirements Model RM in Figure 3.5 is received by the requirements engineer.
- *Performing impact analysis in the Requirements Model RM and Architectural Model AM:* After the change request for requirements is received, the change impact analysis is applied to the RM and AM in Figure 3.5 sequentially. The impacted requirements and architectural elements are identified.

Impacts explosion occurs in change impact analysis in the RM and AM.

- *Updating the Requirements Model RM and Architectural Model AM:* After the impact analysis, it is decided which changes will be applied to the RM and AM. If any, then the RM and AM are modified (New Requirements Model – RM` and New Architectural Model- AM` in Figure 3.5).
- *Verifying the New Architectural Model AM`:* After updating the RM and AM, the new software architecture (New Architectural Model AM`) is verified in order to determine if it satisfies the new/changed requirements.
- *Updating the Architectural Model - AM:* If there is any requirement not satisfied by the New Architectural Model AM`, the AM` is updated to make the new/changed requirements being satisfied.

The software architect might re-analyze the impact of the requirements change in the New Architectural Model AM` to update the AM` where he might encounter impacts explosion.

3.5 Summary of the Problems

The need for change impact analysis is observed in both requirements and software architecture. In the following, we give a summary of the problems in change impact analysis for requirements and software architecture discussed so far. They are tackled in the subsequent chapters.

- **Explosion of Impacts in Requirements for Requirements Changes.** When a change is introduced to a requirement, the requirements engineer needs to find out if any other requirement related to the changed requirement is impacted. In practice, requirements documents are often textual artifacts with implicit structure and analysis of requirements is mostly manual (see *Open Issue 1 – Automation in the early development stages* in Chapter 2). Most of the relations among requirements are not given explicitly. There is a lack of precise definition of relations among requirements in most tools and approaches. Due to the lack of semantics of requirements relations, change impact analysis may produce high number of false positive and false negative impacted requirements. As Bohner stated in [24], semantic information should be employed to overcome the impacts explosion problem (see *Open Issue 2 – Trace Semantics* in Chapter 2). The use of trace semantics reduces the number of false positive impacts. Chapter 4 focusses on the analysis of requirements models for inferencing and consistency checking of requirements relations that are considered as traces. Chapter 4 also provides the formalization of semantics of requirements

relations, which is used to overcome the impacts explosion problem in requirements (see Chapter 5).

- **Manual, Expensive and Error Prone Trace Establishment.** After determining the impacted requirements, the software architect needs to identify impacted architectural elements by tracing the changed requirements to software architecture. Designing architecture based on requirements is a problem solving process that relies on human experience and creativity, and is mainly manual. Therefore, trace information may remain implicit and the software architect may need to manually assign traces between R&A (see *Open Issue 4 – Trace generation from implicit trace information* and *Open Issue 6 – Maintenance of traces* in Chapter 2). Manual trace establishment is time-consuming, expensive and error prone. The assigned traces might be incomplete and invalid. Chapter 6 improves trace establishment between R&A with automation and trace validation.
- **Explosion of Impacts in Software Architecture for Requirements Changes.** There is a lack of precise definition of traces between R&A in most tools and approaches. By using only structural information of traces, the software architect may conclude that all architectural elements in the architecture are impacted. Without considering semantics of traces, change impact analysis may produce high number of false positive impacts in the architecture. Chapter 6 formalizes the semantics of traces between R&A. The semantics is used in Chapter 7 to overcome the impacts explosion problem in software architecture (see *Open Issue 2 – Trace semantics* in Chapter 2).

3.6 Conclusions

In this chapter we answered *Research Question 3* raised in Chapter 1: *What are the change scenarios for requirements and software architecture? What is necessary for these change scenarios to be handled? Which solutions can be used?* We addressed the impacts explosion problem in requirements and software architecture with two change scenarios. Impacts explosion described by Bohner [22] [23] [24] [25] was explained and the specifics of impacts explosion for requirements and software architecture were given. With change scenarios we explained where we might have the impacts explosion for requirements and software architecture.

Bohner [24] states that change impact analysis must employ additional semantic information to increase the accuracy by finding more valid impacts and reducing the number of false-positive impacts. The use of trace semantics in change impact analysis can identify some of

the unimpacted software life-cycle objects at the initial distances and this prevents impact explosion at later distances.

In the thesis we use semantics of requirements and software architecture to prevent impacts explosion. Reasoning about requirements with semantics of requirements relations provided in Chapter 4 supports requirements modeling. Chapter 5 provides a change impact analysis approach to prevent impacts explosion in requirements models. The approach in Chapter 7 uses traces between R&A generated and validated by the approach in Chapter 6 to prevent impacts explosion in software architecture.

Chapter 4

4 Semantics of Requirements Relations

In practice, requirements documents are often textual artifacts with implicit structure. Most of the relations among requirements are not given explicitly. There is a lack of precise definition of relations among requirements in most tools and approaches. In this respect change impact analysis in requirements may produce deficient results. In this chapter, we aim at formal definitions of relation types in order to enable reasoning about requirements relations. We give a requirements metamodel with commonly used relation types. The semantics of the relations is formalized in first-order logic. We use the formalization for consistency checking of relations and for inferring new relations. A tool has been built to support both reasoning activities. We illustrate our approach in an example which shows that the formal semantics of relation types enables new relations to be inferred and contradicting relations in requirements documents to be determined. The results from this chapter are used in Chapter 5 to perform change impact analysis in requirements models.

4.1 Introduction

In Chapter 3, we observed that additional semantic information should be employed to overcome the impacts explosion in requirements and software architecture. Furthermore, there is a need of semantics of requirements and requirements relations to increase the accuracy by finding more valid impacts and reducing the number of false-positive impacts. However, requirements documents are often textual artifacts with structure not explicitly specified. In most tools and approaches there is a lack of precise definition of requirements relations. In this chapter, we aim at identifying requirements relations (see Figure 4.1) and defining their semantics. Within the context of Model Driven Engineering (MDE), we construct metamodels and models for all artifacts in software development. We give a requirements metamodel with formal relation types. The semantics of these relations is based on First-Order Logic (FOL). This *formalization* is used for consistency checking of

relations and inferencing. Here, *inferencing* is the activity of deducing new relations based solely on the relations which the requirements engineer has already specified. *Consistency checking* is the activity of identifying the relations whose existence causes a contradiction. Tool for Requirements Inferencing and Consistency Checking (TRIC) is developed to support both activities. The main features of the tool are managing requirements and relations (add, update, delete), displaying consistency checking and inferencing, and explaining the results of reasoning.

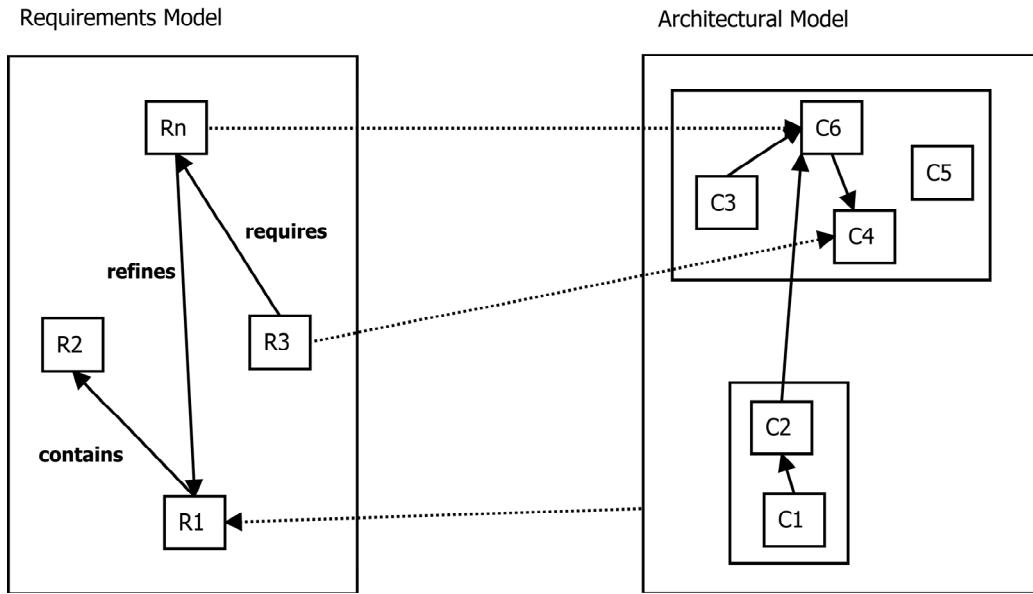


Figure 4.1 Within-Model and Between-Model Traces with Requirements Relation Types for Requirements and Architectural Models

In this chapter we answer *Research Question 4* raised in Chapter 1: *How to model requirements, software architecture and traces with their semantics for change management? What aspects of requirements, software architecture and traces should be modeled and how? How can we use the modeled aspects to reason about requirements, software architecture and traces?* With the requirements metamodel and semantics of requirements relations we address the need for modeling requirements and reasoning about requirements.

Change impact analysis requires semantics of requirements relations which are not given explicitly in most of the approaches. The results in this chapter are used in Chapter 5 to perform change impact analysis in requirements models.

This chapter is structured as follows. Section 4.2 describes the approach. Section 4.3 presents the requirements metamodel and definitions of the requirements relations. The formalization of the relations is provided in Section 4.4. Section 4.5 describes the use of the

formalization for consistency checking and inferencing followed by the details of the tool support in Section 4.6. Section 4.7 illustrates the approach by an example. Section 4.8 describes the related work, and Section 4.9 concludes the chapter.

4.2 Approach

We aim at providing requirements relations with formal semantics. In order to achieve this, we successively take the following steps:

- **Requirements metamodel.** To provide an explicit structure to requirements documents, we define a requirements metamodel. This metamodel includes mostly commonly found entities in the literature. The most important elements of the requirements metamodel are requirements relations and their types (Section 4.3).
- **Semantics of relations.** Since we aim at providing requirements relations with well-defined semantics, we formalize the requirements relations by using FOL (Section 4.4).
- **Consistency checking and inferencing.** We use the formalization for consistency checking of relations and inferring new relations (Section 4.5).
- **Tool support.** We describe the design and implementation of a tool for managing requirements, displaying consistency checking & inferencing, and explaining results of reasoning (Section 4.6).
- **Running example.** We illustrate the approach with an example (Section 4.7). The example is about requirements for a Course Management System (CMS). This system provides a lecturer with a set of tools that allows the creation of online course content and the subsequent teaching and management of that course including interactions with students taking the course. A CMS requirements document was put together for illustration in this chapter as a running example. Part of this document is given in Appendix B.

4.3 Requirements Metamodel

Our requirements metamodel contains common entities identified in the literature for requirements models. There are several commonly used approaches to define and represent requirements: goal-oriented [250] [186], aspect-driven [216], variability management [183], use-case [54], domain-specific [200] [143], and reuse-driven techniques [164]. Goal-oriented

requirements engineering [250] [186] defines a model for decomposing a system goal into requirements with goal trees, and offers some decision methods based on this decomposition. The aspect-oriented approach [216] gives a requirements model for the separation of crosscutting functional and non-functional properties in the requirements analysis phase. The System Modeling Language (SysML) [200] is a domain-specific modeling language for system engineering. It provides modeling constructs to represent text-based requirements and relate them to other modeling elements with stereotypes. The variability management approach [183] deals with producing requirements that can be considered as a core asset in a product line.

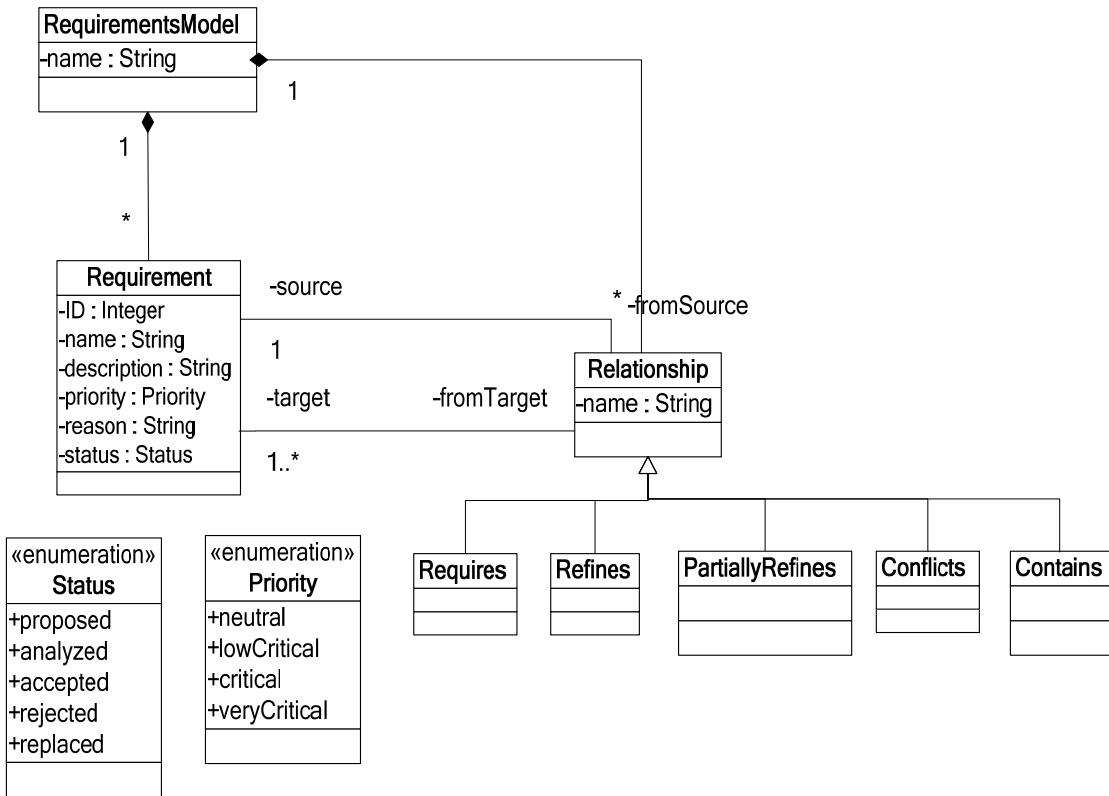


Figure 4.2 Requirements Metamodel

Since we aim at using requirements relations as trace relations, we focused in our survey on the requirement entity with its attributes and relations between requirements. We left out other entities such as goals, stakeholders, and test cases. Figure 4.2 gives the requirements metamodel used in our approach.

In the requirements metamodel, requirements are captured in a *requirements model*. A requirements model contains *requirements* and their *relationships*. Based on [239], we define a requirement as follows:

- **Definition 4.1.** *Requirement:* A requirement is a description of a system property or properties which need to be fulfilled.

Requirements relations are defined as follows:

- **Definition 4.2.** *Requires relation:* A requirement R_1 requires a requirement R_2 if R_1 is fulfilled only when R_2 is fulfilled.

The required requirement can be seen as a pre-condition for the requiring requirement [255].

- **Definition 4.3.** *Refines relation:* A requirement R_1 refines a requirement R_2 if R_1 is derived from R_2 by adding more details to its properties.

The refined requirement can be seen as an abstraction of the detailed requirements [59] [250].

- **Definition 4.4.** *Partially refines relation:* A requirement R_1 partially refines a requirement R_2 if R_1 is derived from R_2 by adding more details to properties of R_2 and excluding the unrefined properties of R_2 .

Our assumption here is that R_2 can be decomposed into other requirements and that R_1 refines a subset of these decomposed requirements. This relation can be described as a special combination of decomposition and refinement. It is mainly drawn from the decomposition of goals in goal-oriented requirements engineering [250].

- **Definition 4.5.** *Contains relation:* A requirement R_1 contains requirements $R_2 \dots R_n$ if $R_2 \dots R_n$ are parts of the whole R_1 (part-whole hierarchy).

This relationship enables a complex requirement to be decomposed into parts [200]. 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. For this relation, all parts are required in order to fulfill the composing requirement.

- **Definition 4.6.** *Conflicts relation:* 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.

The *conflicts* relation addresses a contradiction between requirements. This relation may be modeled explicitly by the requirements engineer. In this thesis, we consider *conflicts* as a binary relation [251]. Our approach can be extended to n-ary conflicts relations, that is,

conflicts among multiple requirements, as a whole without excluding pairs of requirements to be fulfilled.

The conflicts relation should be distinguished from inconsistencies in requirements relations. In our terminology, an *inconsistency* is a situation where the co-existence of certain relations among requirements causes a contradiction in the context of the semantics given in this chapter. When we use the term *consistency checking*, we refer to finding inconsistencies among requirements relations (more on this in Section 4.5).

There are other classifications of inconsistencies between requirements. For example, Van Lamsweerde et al. [251] distinguish *conflicts* (excluding the simultaneous fulfillment of requirements), *divergence* (boundary cases make requirements contradict – a weaker form of conflict), *competition* (a particular case of divergence), *obstruction* (a borderline case of divergence), and *terminology clash* (using different syntactic names for a single real-world concept).

The definitions given above are informal and present an intuitive meaning (and sometimes ambiguous). Since we aim at precise semantics, we formalize requirements and requirements relations in FOL.

4.4 Formalization of Requirements and Relations

In this section we provide formalization of requirements and relation types. Section 4.4.1 gives the formalization of requirements. Section 4.4.2 presents the formalization of requirements relations. We chose a formalization of requirements in first-order logic (FOL). We discuss this choice in Section 4.4.3.

4.4.1 Formalization of Requirements

We assume the general notion of requirement being “a property which must be exhibited by a system”. We express the property as a formula P in FOL. We assume that requirements can always be expressed in the universal fragment of FOL and a requirement is expressed as a formula $\forall x\varphi$ with φ in conjunctive normal form (CNF). If the formula φ is a closed formula, then the universal quantifiers can be dropped. It is also possible that the formula contains free variables.

According to the model theoretic semantics of FOL the truth value of P is determined in a *model* \mathcal{M} by using an interpretation for the function and predicate symbols in P .

Let \mathcal{F} be a set of function symbols and \mathcal{P} a set of predicate symbols, each symbol with a fixed arity. A model \mathcal{M} of the pair $(\mathcal{F}, \mathcal{P})$ consists of the following items [118]:

- a non-empty set A , the *universe of concrete values*
- for each $f \in \mathcal{F}$ with n arguments, a function $f^{\mathcal{M}} : A^n \rightarrow A$
- for each $P \in \mathcal{P}$ with n arguments, a set $P^{\mathcal{M}} \subseteq A^n$.

The details of the definition of a model in FOL can be found in Appendix A. A satisfaction relation between the model \mathcal{M} and the formula P holds:

$$(1) \quad \mathcal{M} \models_{\ell} P$$

if P evaluates to True in the model \mathcal{M} with respect to the environment ℓ (i.e., a look-up table for free variables in P). The model \mathcal{M} together with ℓ in which P is true represents a system s that satisfies the requirement. From now on, all the formulae P that express properties will be in the form where $(\forall x = \forall x_1 \ \forall x_2 \ \dots \ \forall x_k)$:

$$(2) \quad P = \forall x \ (p_1 \wedge \dots \wedge p_n), \text{ where } n \geq 1$$

p_n is a disjunction of literals which are atomic formulas (atoms) or their negation. An atomic formula is a predicate symbol applied over terms. In the rest of the thesis we use the notation $(p_1 \dots p_n)$ for $(p_1 \wedge \dots \wedge p_n)$.

Example: Interpretation of a Requirement as a Formula

Although the interpretation of requirements as formulas in FOL is not within the scope of our work, we give an intuition of how to map requirements expressed in natural text to our formalization in FOL. Assume that we have the following requirement: “*The system shall provide security facilities for login*”.

We can represent the requirement as a formula $provide(x; \text{login}) \wedge \text{security_mechanism}(x)$. The intuition behind x is that x is a free variable ranging over possible security solutions (since security can be supported in different ways, e.g. SSL certification, TLS certification). login is a constant. An example system for this requirement supports SSL certification for users to log in.

Let $\mathcal{F} \triangleq \{\text{login}\}$ and $\mathcal{P} \triangleq \{\text{provide}, \text{security_mechanism}\}$, where login is a constant symbol; and where provide is a predicate with two arguments and $\text{security_ mechanism}$ is a predicate with just one argument. We choose as a model \mathcal{M} the following:

- $\mathbf{A} \triangleq \{\text{ssl_certification}, \text{tls_certification}, \text{socket_communication}, \text{login_feature}\}$
- $\text{login}^{\mathcal{M}} \triangleq \text{login_feature}$
- $\text{provide}^{\mathcal{M}} \triangleq \{(\text{ssl_certification}, \text{login_feature})\}$
- $\text{security_ mechanism}^{\mathcal{M}} \triangleq \{\text{ssl_certification}, \text{tls_certification}\}$

We have the following satisfaction relation between the model \mathcal{M} and the formula stated in the requirement:

$$(3) \quad \mathcal{M} \models \ell_{[x \mapsto \text{ssl_certification}]} \text{provide}(x, \text{login}) \wedge \text{security_ mechanism}(x)$$

where ℓ maps the free variable x to the value ssl_certification in the set \mathbf{A} and login is the constant. The model \mathcal{M} together with ℓ can be considered as a part of the system s that satisfies the requirement.

The model may express both modeling choices and universal truths (domain knowledge). The relation $\text{security_ mechanism}^{\mathcal{M}}$ refers to a universal truth for available security mechanisms. The relation $\text{provide}^{\mathcal{M}}$ refers to modeling choices like providing login feature with ssl certification. In a different model, login feature might be provided with tls certification or just with an unsecure socket communication. Consider the relation $\text{provide}^{\mathcal{M}'}$ in the model \mathcal{M}' (\mathbf{A} , $\text{login}^{\mathcal{M}'}$ and $\text{security_ mechanism}^{\mathcal{M}'}$ are kept same with the model \mathcal{M}):

- $\text{provide}^{\mathcal{M}'} \triangleq \{(\text{socket_communication}, \text{login_feature})\}$

The formula is not satisfied in the model \mathcal{M}' and the environment ℓ which maps the variable x to the value $\text{socket_communication}$ in the set \mathbf{A} .

$$(4) \quad \mathcal{M}' \not\models \ell_{[x \mapsto \text{socket_communication}]} \text{provide}(x, \text{login}) \wedge \text{security_ mechanism}(x)$$

The system provides a login feature with an unsecure socket communication.

4.4.2 Formalization of Requirements Relations

We formalize the informal definitions of the requirements relations in the requirements metamodel.

4.4.2.1 Formalization of Requires

Let R_1 and R_2 be requirements where P_1 and P_2 are formulas in CNF for R_1 and R_2 .

R_1 requires R_2 iff the following two statements hold:

$$(5) \quad (P_1 \rightarrow P_2)$$

$$(6) \quad (\neg(P_2 \rightarrow P_1)) \text{ is satisfiable}$$

Please note that if the requirements R_1 and R_2 are written as formulas $\forall x\varphi$ and $\forall x\psi$ with φ and ψ in CNF, we understand the following: R_1 requires R_2 iff $(\forall x(\varphi \rightarrow \psi))$ and $(\neg(\forall x(\psi \rightarrow \varphi)))$ is satisfiable. This is also valid for other relations.

From the definition we conclude that $(S_1 \subset S_2)$ where S_1 is the set of systems that satisfy R_1 and S_2 is the set of systems that satisfy R_2 . The *requires* relation is *non-reflexive*, *non-symmetric*, and *transitive*.

Example: Requires Relation

We explain the *requires* relation with the following two requirements from the CMS requirements document explained in Section 4.7.

R24: The system shall notify students about events (new messages posted, etc.).

R7: The system shall provide a messaging facility.

We formalize the requirements R7 and R24 as follows:

$$(7) \quad P_7 = \text{provide_msg}(x)$$

$$(8) \quad P_{24} = \text{notify_students}(x, \text{std_events})$$

where x is a free variable over the values in A and std_events is a constant. Let $\mathcal{F} \equiv \{\text{std_events}\}$ and $\mathcal{P} \equiv \{\text{provide_msg}, \text{notify_students}\}$, where std_events is a constant symbol; and where *provide_msg* is a predicate with one argument and *notify_students* is a predicate with two arguments. From the domain knowledge we know that for all models if $((x, \text{std_events}) \in \text{notify_students}^M)$, then $(x \in \text{provide_msg}^M)$. We choose as a model M the following:

- $A \equiv \{\text{individual_msg}, \text{team_msg}, \text{participant_msg}, \text{lecturer_msg}, \text{student_events}\}$
- $\text{std_events}^M \equiv \text{student_events}$

- $\text{provide_msg}^{\mathcal{M}} \stackrel{\text{def}}{=} \{\text{individual_msg}, \text{team_msg}, \text{participant_msg}, \text{lecturer_msg}\}$
- $\text{notify_students}^{\mathcal{M}} \stackrel{\text{def}}{=} \{(\text{individual_msg}, \text{student_events}), (\text{team_msg}, \text{student_events}), (\text{participant_msg}, \text{student_events})\}$

Then we have the following:

$$(9) \quad \mathcal{M} \models \ell \text{ notify_students}(x, \text{std_events}) \rightarrow \text{provide_msg}(x)$$

$(\text{notify_students}(x, \text{std_events}) \rightarrow \text{provide_msg}(x))$ holds for all bindings of x in the environment ℓ since for all models if $((x, \text{std_events}) \in \text{notify_students}^{\mathcal{M}})$, then $(x \in \text{provide_msg}^{\mathcal{M}})$. $(\neg(\text{provide_msg}(x) \rightarrow \text{notify_students}(x, \text{std_events})))$ is satisfiable:

$$(10) \quad \mathcal{M} \models \ell[x \mapsto \text{lecturer_msg}] (\neg(\text{provide_msg}(x) \rightarrow \text{notify_students}(x, \text{std_events})))$$

Therefore, we conclude that R24 requires R7 to be fulfilled.

4.4.2.2 Formalization of Refines

Let R_1 and R_2 be requirements. P_1 and P_2 are formulas for R_1 and R_2 . The conjunctive normal form of P_2 is:

$$(11) \quad P_2 = \forall x ((p_1 \dots p_n) \wedge (q_1 \dots q_m)); n \geq 1, m \geq 0$$

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

R_1 refines R_2 iff P_1 is derived from P_2 by replacing every p_j in P_2 with p_j^l for $j \in 1..n$ such that the following two statements hold:

$$(12) \quad P_1 = \forall x ((p_1^l \dots p_n^l) \wedge (q_1 \dots q_m)); n \geq 1, m \geq 0$$

$$(13) \quad (\neg(\forall x (p_j \rightarrow p_j^l))) \text{ is satisfiable for all } j \in 1..n$$

From the definition we conclude that $(P_1 \rightarrow P_2)$ holds for every model where R_1 refines R_2 and $(\neg(P_2 \rightarrow P_1))$ is satisfiable. Therefore, $(S_1 \subset S_2)$ where S_1 is the set of systems that satisfy R_1 and S_2 is the set of systems that satisfy R_2 . Similarly to the previous relation we have the properties *non-reflexive*, *non-symmetric*, and *transitive* for the *refines* relation. Obviously, if R_1 refines R_2 then R_1 requires R_2 .

Example: Refines Relation

We explain the *refines* relation with the following two requirements.

R7: The system shall provide a messaging facility.

R16: The system shall allow messages to be sent to individuals, teams, or all course participants at once.

We formalize the requirements R7 and R16 as follows:

$$(14) \quad P_7 = \text{provide_msg}(x)$$

$$(15) \quad P_{16} = \text{course_msg}(x)$$

where x is a free variable over the values in A . Let $\mathcal{P} \stackrel{\text{def}}{=} \{\text{provide_msg}, \text{course_msg}\}$ where *provide_msg* and *course_msg* are predicates with one argument. From the domain knowledge we are interested only in those models for which (16) holds:

$$(16) \quad \text{course_msg}^M \subseteq \text{provide_msg}^M$$

We choose as a model M the following:

- $A \stackrel{\text{def}}{=} \{\text{individual_msg}, \text{team_msg}, \text{participant_msg}, \text{lecturer_msg}\}$
- $\text{provide_msg}^M \stackrel{\text{def}}{=} \{\text{individual_msg}, \text{team_msg}, \text{participant_msg}, \text{lecturer_msg}\}$
- $\text{course_msg}^M \stackrel{\text{def}}{=} \{\text{individual_msg}, \text{team_msg}, \text{participant_msg}\}$

Then we have the following:

$$(17) \quad M \models \ell \text{course_msg}(x) \rightarrow \text{provide_msg}(x)$$

$(\text{course_msg}(x) \rightarrow \text{provide_msg}(x))$ holds for each model M since $(\text{course_msg}^M \subseteq \text{provide_msg}^M)$ for all models. $(\neg(\text{provide_msg}(x) \rightarrow \text{course_msg}(x)))$ is satisfiable like in the following:

$$(18) \quad M \models \ell_{[x \mapsto \text{lecturer_msg}]} (\neg(\text{provide_msg}(x) \rightarrow \text{course_msg}(x)))$$

R7 states only the need for a messaging property in the system. However, R16 explains the details of the messaging property: the messaging shall allow messages to be sent to individuals, teams, or all course participants at once, excluding lecturers. Therefore, we conclude that R16 refines R7. Note also that R16 requires R7 to be fulfilled.

4.4.2.3 Formalization of Partially Refines

Let R_1 and R_2 be requirements. P_1 and P_2 are formulas for R_1 and R_2 . The conjunctive normal form of P_2 is:

$$(19) \quad P_2 = \forall x ((p_1 \dots p_n) \wedge (q_1 \dots q_m)); \quad m, n \geq 1$$

Let $q_1^l, q_2^l, \dots, q_{m-1}^l, q_m^l$ be disjunction of literals such that $\forall x (q_i^l \rightarrow q_i)$ for all $i \in 1..m$

R_1 partially refines R_2 iff P_1 is derived from P_2 by replacing every q_i in P_2 with q_i^l for $i \in 1..m$ and excluding others (p_i for all $i \in 1..n$) such that the following two statements hold:

$$(20) \quad P_1 = \forall x (q_1^l \dots q_m^l)$$

$$(21) \quad (\neg(\forall x (q_i \rightarrow q_i^l))) \text{ is satisfiable for all } i \in 1..m$$

The *partially refines* relation is *non-reflexive*, *non-symmetric*, and *transitive*.

Example: Partially Refines Relation

We explain the *partially refines* relation with the following two requirements.

R97: The system shall allow only the administration to *manage* courses.

R102: The system shall allow only the administration to specify the minimum number of students for a course. If there are too few subscriptions in a semester, that course will not be given during that semester.

In the glossary of the CMS requirements document in Appendix B, it is stated that *managing courses* means *creating*, *updating*, *deleting*, and *reading course information*. We formalize R97 and R102 as follows:

$$(22) \quad P_{97} = \forall x \forall y ((\neg \text{courses}(x) \vee \neg \text{numbers}(y) \vee \text{allow_admin_create}(x, y)) \wedge (\neg \text{courses}(x) \vee \neg \text{numbers}(y) \vee \text{allow_admin_delete}(x, y)) \wedge (\neg \text{courses}(x) \vee \neg \text{numbers}(y) \vee \text{allow_admin_update}(x, y)) \wedge (\neg \text{courses}(x) \vee \neg \text{numbers}(y) \vee \text{allow_admin_read}(x, y)))$$

$$(23) \quad P_{102} = \forall x \forall y (\neg \text{courses}(x) \vee \neg \text{numbers}(y) \vee \text{allow_admin_specify}(x, y, z))$$

where x is a universally quantified variable for the courses, y is a universally quantified variable for the number of students registered to the course and z is a free variable for the minimum number of students that should be registered to the course.

Let $\mathcal{P} \triangleq \{\text{allow_admin_create}, \text{allow_admin_delete}, \text{allow_admin_update}, \text{allow_admin_read}, \text{allow_admin_specify}, \text{courses}, \text{numbers}\}$ where *courses* and *numbers* are predicates with one argument, *allow_admin_create*, *allow_admin_delete*, *allow_admin_update* and *allow_admin_read* are predicates with two arguments, and *allow_admin_specify* is a predicate with three arguments. From the domain knowledge we are interested only in those models that satisfy the following condition: If $((x, y, z) \in \text{allow_admin_specify}^M)$, then $((x, y) \in \text{allow_admin_create}^M)$.

We choose as a model M the following:

- The universe of concrete values A consists of the elements that correspond to the courses and the positive natural numbers. The elements for the courses are *mathematics*, *physics*, *chemistry*, *biology* and *literature*.
- $\text{allow_admin_create}^M \triangleq \{(x, y) \mid x \in \{\text{mathematics}, \text{physics}, \text{chemistry}, \text{biology}, \text{literature}\}, y \in \mathbb{N}^+\}$
- $\text{allow_admin_delete}^M \triangleq \{(x, y) \mid x \in \{\text{mathematics}, \text{physics}, \text{chemistry}, \text{biology}, \text{literature}\}, y \in \mathbb{N}^+\}$
- $\text{allow_admin_update}^M \triangleq \{(x, y) \mid x \in \{\text{mathematics}, \text{physics}, \text{chemistry}, \text{biology}, \text{literature}\}, y \in \mathbb{N}^+\}$
- $\text{allow_admin_read}^M \triangleq \{(x, y) \mid x \in \{\text{mathematics}, \text{physics}, \text{chemistry}, \text{biology}, \text{literature}\}, y \in \mathbb{N}^+\}$
- $\text{allow_admin_specify}^M \triangleq \{(x, y, z) \mid (x, y) \in \text{allow_admin_create}^M, z \in \mathbb{N}^+, y \geq z\}$
- $\text{courses}^M \triangleq \{x \mid x \in \{\text{mathematics}, \text{physics}, \text{chemistry}, \text{biology}, \text{literature}\}\}$
- $\text{numbers}^M \triangleq \{y \mid y \in \mathbb{N}^+\}$

We interpret P₁₀₂ as assigning the minimum number of students for a created course. Then we have the following:

$$(24) \quad M \models \ell \forall x \forall y ((\neg \text{courses}(x) \vee \neg \text{numbers}(y) \vee \text{allow_admin_specify}(x, y, z)) \rightarrow (\neg \text{courses}(x) \vee \neg \text{numbers}(y) \vee \text{allow_admin_create}(x, y)))$$

The formula holds for each model since for all models if $((x, y, z) \in \text{allow_admin_specify}^M)$, then $((x, y) \in \text{allow_admin_create}^M)$.

$(\neg(\forall x \forall y ((\neg\text{courses}(x) \vee \neg\text{numbers}(y) \vee \text{allow_admin_create}(x, y)) \rightarrow (\neg\text{courses}(x) \vee \neg\text{numbers}(y) \vee \text{allow_admin_specify}(x, y, z))))$ is satisfiable:

$$(25) \quad M \models \ell (\neg(\forall x \forall y ((\neg\text{courses}(x) \vee \neg\text{numbers}(y) \vee \text{allow_admin_create}(x, y)) \rightarrow (\neg\text{courses}(x) \vee \neg\text{numbers}(y) \vee \text{allow_admin_specify}(x, y, z))))$$

There is at least a value of the variable z where $y < z$. Therefore, R102 partially refines R97.

4.4.2.4 Formalization of Contains

Let R_1, R_2, \dots, R_k be requirements where $k \geq 2$. $P_1, P_2, P_3, \dots, P_k$ are formulas for R_1, R_2, \dots, R_k in conjunctive normal form as follows:

$$(26) \quad P_i = \forall x (p_{1i} \dots p_{mi}) ; m_i \geq 1, i \in 2 \dots k$$

R_1 contains R_2, \dots, R_k iff P_1 is derived from P_2, P_3, \dots, P_k such that the following two statements hold:

$$(27) \quad P_1 = P_2 \wedge P_3 \wedge \dots \wedge P_k \wedge P^l$$

$$(28) \quad (\neg(P_2 \rightarrow P_1)), (\neg(P_3 \rightarrow P_1)), \dots, (\neg(P_k \rightarrow P_1)) \text{ are satisfiable}$$

where P^l denotes properties that are not captured in P_2, P_3, \dots, P_k

For the formulas P_1, P_2, \dots, P_k , if any variable universally quantified in one of the formulas appears free in any other formulas, the free variable is renamed. If any variable in one of the formulas appears in any other formulas with a different valuation, the variable with the different valuation is renamed. Please note that if the requirements R_1, R_2, \dots, R_k are written as formulas $\forall x \varphi_1, \forall x \varphi_2, \dots, \forall x \varphi_k$ with $\varphi_1, \varphi_2, \dots, \varphi_k$ in CNF and P^l is expressed as $\forall x \psi$ with ψ in CNF, we understand the following: R_1 contains R_2, \dots, R_k iff $(P_1 = \forall x (\varphi_2 \wedge \varphi_3 \wedge \dots \wedge \varphi_k \wedge \psi))$ and $(\neg(\forall x (\varphi_2 \rightarrow \varphi_1))), (\neg(\forall x (\varphi_3 \rightarrow \varphi_1))), \dots, (\neg(\forall x (\varphi_k \rightarrow \varphi_1)))$ are satisfiable.

In the definition, we do not assume completeness of the decomposition [250]. From the definition we conclude that $(P_1 \rightarrow P_2), (P_1 \rightarrow P_3), \dots$, and $(P_1 \rightarrow P_k)$ hold for every model where R_1 contains R_2, \dots, R_k . Therefore, $S_1 \subset S_2, S_1 \subset S_3, \dots$, and $S_1 \subset S_k$ where S_1, S_2, S_3, \dots ,

and S_k are the set of systems that satisfy R_1, R_2, R_3, \dots , and R_k . The *contains* relation is *non-reflexive*, *non-symmetric*, and *transitive*. Obviously, if R_1 contains R_2 then R_1 requires R_2 .

Example: Contains Relation

We explain the *contains* relation with the following two requirements.

R61: The system shall allow lecturers to specify enrolment policies based on grade, first-come first-serve (fcfs), and department.

R62: The system shall allow lecturers to specify enrolment policies based on grade.

We formalize R61 and R62 as follows

$$(29) \quad P_{61} = \text{allow_policy}(\text{grade_enrl_policy}) \wedge \text{allow_policy}(\text{fcfs_enrl_policy}) \wedge \text{allow_policy}(\text{department_enrl_policy})$$

$$(30) \quad P_{62} = \text{allow_policy}(\text{grade_enrl_policy})$$

where *grade_enrl_policy*, *fcfs_enrl_policy*, and *department_enrl_policy* are constants. We have the following:

$$(31) \quad P_{61} = P_{62} \wedge \text{allow_policy}(\text{fcfs_enrl_policy}) \wedge \text{allow_policy}(\text{department_enrl_policy})$$

Let $\mathcal{F} \triangleq \{\text{fcfs_enrl_policy}, \text{department_enrl_policy}, \text{grade_enrl_policy}\}$ and $\mathcal{P} \triangleq \{\text{allow_policy}\}$, where *fcfs_enrl_policy*, *department_enrl_policy* and *grade_enrl_policy* are constant symbols; and where *allow_policy* is a predicate with one argument. We choose as a model \mathcal{M} the following:

- $A \triangleq \{\text{fcfs_enrolment_policy}, \text{department_enrolment_policy}, \text{grade_enrolment_policy}\}$
- $\text{fcfs_enrl_policy}^{\mathcal{M}} \triangleq \text{fcfs_enrolment_policy}$
- $\text{department_enrl_policy}^{\mathcal{M}} \triangleq \text{department_enrolment_policy}$
- $\text{grade_enrl_policy}^{\mathcal{M}} \triangleq \text{grade_enrolment_policy}$
- $\text{allow_policy}^{\mathcal{M}} \triangleq \{\text{fcfs_enrolment_policy}, \text{grade_enrolment_policy}\}$

Then we have the following:

$$(32) \quad \mathcal{M} \models \neg(\text{allow_policy}(\text{grade_enrl_policy}) \rightarrow (\text{allow_policy}(\text{grade_enrl_policy}) \wedge \text{allow_policy}(\text{fcfs_enrl_policy}) \wedge \text{allow_policy}(\text{department_enrl_policy})))$$

R61 states that the system shall allow lecturers to specify three different enrollment policies. The requirement can be interpreted as three different properties for the system, like *specifying enrollment policies based on grade*, *specifying enrollment policies based on first come first serve*, and *specifying enrollment policies based on department*. R62 states only one of these properties, which is *specifying enrollment policies based on grade*. Therefore, we conclude that R62 is one of the decomposed requirements of R61 (R61 contains R62). It is also noted that R61 requires R62 to be fulfilled.

4.4.2.5 Formalization of Conflicts

Let R_1 and R_2 be requirements. P_1 and P_2 are formulas for R_1 and R_2 .

R_1 conflicts with R_2 iff $(P_1 \rightarrow \neg P_2) \wedge (P_2 \rightarrow \neg P_1)$

Please note that if the requirements R_1 and R_2 are written as formulas $\forall x \varphi$ and $\forall x \psi$ with φ and ψ in CNF, we understand the following: R_1 conflicts R_2 iff $(\forall x (\varphi \rightarrow \neg \psi)) \wedge (\forall x (\psi \rightarrow \neg \varphi))$.

From the definition we conclude that $(S_1 \cap S_2) = \emptyset$ where S_1 is the set of systems that satisfy R_1 and S_2 is the set of systems that satisfy R_2 . The binary *conflicts* relation is *symmetric* and *non-reflexive*. It is not *transitive*.

Example: Conflicts Relation

We explain the *conflicts* relation with the following two requirements.

R60: The system shall allow lecturers to limit the number of students subscribing to a course.

R103: The system shall have no maximum limit on the number of course participants ever.

We formalize R60 and R103 as follows:

$$(33) \quad P_{60} = \forall x \forall y (\neg \text{courses}(x) \vee \neg \text{numbers}(y) \vee \text{allow_lecturer_limit}(x, y))$$

$$(34) \quad P_{103} = \forall x \forall y (\neg \text{courses}(x) \vee \neg \text{numbers}(y) \vee \neg \text{has_limit}(x, y))$$

where x is a universally quantified variable for the courses and y is a universally quantified variable for the limit of the number of students that should be registered to the course.

Let $\mathcal{P} \stackrel{\text{def}}{=} \{\text{allow_lecturer_limit}, \text{has_limit}, \text{courses}, \text{numbers}\}$ where *courses* and *numbers* are predicates with one argument, *allow_lecturer_limit* and *has_limit* are predicates with two

arguments. From the domain knowledge we are interested only in those models where the following statement is valid:

$$(35) \quad \text{allow_lecturer_limit}^{\mathcal{M}} = \text{has_limit}^{\mathcal{M}}$$

We choose as a model \mathcal{M} the following:

- The universe of concrete values A consists of the elements that correspond to the courses and the positive natural numbers. The elements for the courses are *mathematics*, *physics*, *chemistry*, *biology* and *literature*.
- $\text{allow_lecturer_limit}^{\mathcal{M}} \stackrel{\text{def}}{=} \{(x, y) \mid x \in \{\text{mathematics, physics, chemistry, biology, literature}\}, y \in \mathbb{N}^+\}$
- $\text{has_limit}^{\mathcal{M}} \stackrel{\text{def}}{=} \{(x, y) \mid x \in \{\text{mathematics, physics, chemistry, biology, literature}\}, y \in \mathbb{N}^+\}$
- $\text{courses}^{\mathcal{M}} \stackrel{\text{def}}{=} \{x \mid x \in \{\text{mathematics, physics, chemistry, biology, literature}\}\}$
- $\text{numbers}^{\mathcal{M}} \stackrel{\text{def}}{=} \{y \mid y \in \mathbb{N}^+\}$

Then we have the following:

$$(36) \quad \mathcal{M} \models \ell \forall x \forall y ((\neg \text{courses}(x) \vee \neg \text{numbers}(y) \vee \text{allow_lecturer_limit}(x, y)) \rightarrow (\neg(\neg \text{courses}(x) \vee \neg \text{numbers}(y) \vee \neg \text{has_limit}(x, y))))$$

$$(37) \quad \mathcal{M} \models \ell \forall x \forall y ((\neg \text{courses}(x) \vee \neg \text{numbers}(y) \vee \neg \text{has_limit}(x, y)) \rightarrow (\neg(\neg \text{courses}(x) \vee \neg \text{numbers}(y) \vee \text{allow_lecturer_limit}(x, y))))$$

The formulas in (36) and (37) hold for each model since $(\text{allow_lecturer_limit}^{\mathcal{M}} = \text{has_limit}^{\mathcal{M}})$ for all models. The satisfaction of R60 excludes the satisfaction of R103 and vice versa. The limit on the number of students and absence of a maximum limit on the number of course participants cannot exist at the same time. Therefore, we conclude that R60 conflicts with R103.

4.4.3 Discussion on the Chosen Formalization

We chose a formalization of requirements and their relations in FOL. There are other formalizations of requirements, for example, in modal logic and deontic logic [177]. The formalization in FOL allows the expression of commonly occurring requirement

descriptions. However, there are limitations of the expressivity of FOL. For instance, imperfect requirements can be modeled by fuzzy sets [191]. Dealing with imperfection is out of scope of our formalization. We also do not cover modalities in requirements like possibility, probability, and necessity or logic operators like “in the next state” and “some time in the future” which can be used to describe the evolution of requirements. Our formalization should be extended with temporal logic, modal logic or fuzzy sets in order to cover these types of requirements. Under these limitations, the expressiveness of FOL is sufficient for inferencing and consistency checking.

As we stated in Section 4.4.1, the interpretation of requirements as formulas in FOL is not within the scope of our approach. The modeling of requirements and their relations is carried out by requirements engineers. However, the requirements engineer does not need to know the details of the formalization. He/she can be guided by tutorials [94] that provide an informal explanation of the relations. The requirements model is used to obtain new knowledge about the requirements relations by automated reasoning, for example, inferred relations and/or inconsistencies. These results – supported by the visualization – are presented to the requirements engineer, who should give his/her own interpretation. Since the requirements engineer may make mistakes in the modeling, the approach may produce incorrect results. However, by interpreting the results, the requirements engineer may improve his initial requirements model. Also the consistency checking may detect errors.

4.5 Inferencing and Consistency Checking

Inferencing and consistency checking aim at deriving new relations based on given relations and determining contradictions among relations. Rules can be derived for the combinations of relations where new relations are inferred and contradictions are determined. Some of the inference rules are as follows:

- $(R_1 \text{ refines } R_2) \wedge (R_2 \text{ contains } R_3) \rightarrow (R_1 \text{ requires } R_3)$
- $(R_1 \text{ refines } R_2) \wedge (R_2 \text{ contains } R_3) \rightarrow (R_1 \text{ requires } R_3)$
- $(R_1 \text{ partially-refines } R_2) \wedge (R_1 \text{ contains } R_3) \rightarrow (R_3 \text{ partially-refines } R_2)$
- $(R_1 \text{ contains } R_2) \rightarrow (R_1 \text{ requires } R_2)$
- $(R_1 \text{ refines } R_2) \rightarrow \neg(R_2 \text{ requires } R_1)$

It is not easy to ensure the completeness of the rules. Instead of exploring combinations of requirements relations, requirements relations are represented as facts derived from their

definitions. The inferencing and consistency checking are implemented in a reasoner supporting a form of logic programming based on these facts. The first type of facts concerns relations among sets, and the second encodes relations between formulas. Transitive and disjoint properties of the set and formula relations are used together with some inferencing rules. The formula relations are defined for formulas in CNF.

Since the rules of set theory and formula relations can be directly mapped to the Web Ontology Language (OWL) [62], we use an OWL reasoner called Jena [130] in our implementation. The Web Ontology Language (OWL) is a family of knowledge representation languages for specifying ontologies. OWL ontologies are serialized using RDF/XML syntax. Our formalization is directly mapped to the language features of OWL like transitivity and symmetry of properties. Reasoning on requirements models is done on OWL ontologies. We used Jena [130], a programmatic environment for processing OWL data, with a rule-based inference engine. The engine performs consistency checking and inferencing. The details of the tool support with OWL for inferencing and consistency checking are given in Section 4.6. In the following, we illustrate how to map requirements relations to facts and the use of the facts for inferencing and consistency checking.

Mapping Requirements Relations to Set Theoretic Relations. The set theoretic relations with their properties like transitivity are natively supported in OWL and OWL reasoners. Therefore, based on the formalization of the relations, we map the requirements relations (*requires*, *refines*, *contains*, and *conflicts*) to the set theoretic relations for the set of systems.

Let R_1 and R_2 be requirements. P_1 and P_2 are formulas for R_1 and R_2 . S_1 is the set of systems that satisfy R_1 and S_2 is the set of systems that satisfy R_2 .

- $(S_1 \subset S_2)$ iff $(R_1 \text{ requires } R_2)$
- $(S_1 \subset S_2)$ if $(R_1 \text{ refines } R_2)$
- $(S_1 \subset S_2)$ if $(R_1 \text{ contains } R_2)$
- $((S_1 \cap S_2) = \emptyset)$ iff $(R_1 \text{ conflicts } R_2)$

To map the partially refines relation to the set theoretic relations for a set of systems, we decompose this relation to the combination of contains and refines relations. We define a temporary requirement named R_{T12} to decompose the partially refines relation between R_1 and R_2 into refines and contains relations. The partially refines relation can be decomposed into the contains and refines relations in two different combinations:

- $(R_2 \text{ contains } R_{T12}) \wedge (R_1 \text{ refines } R_{T12}) \text{ iff } (R_1 \text{ partially-refines } R_2)$
- $(R_{T12} \text{ refines } R_2) \wedge (R_{T12} \text{ contains } R_1) \text{ iff } (R_1 \text{ partially-refines } R_2)$

The combinations given above exist at the same time. Each combination is mapped to set theoretic relations.

- $(S_2 \subset S_{T12}) \wedge (S_1 \subset S_{T12}) \text{ if } (R_1 \text{ partially-refines } R_2)$
- $(S_{T12} \subset S_2) \wedge (S_{T12} \subset S_1) \text{ if } (R_1 \text{ partially-refines } R_2)$

The mappings are implemented by using the Jena reasoner rule language. The Jena rules for the mappings can be found in Appendix C. An informal description of the simplified text rule syntax [130] is:

```

Rule    := bare-rule .
         or [ bare-rule ]
         or [ ruleName : bare-rule ]

bare-rule := term, ... term -> hterm, ... hterm // forward rule
           or bhterm <- term, ... term           // backward rule

hterm   := term
         or [ bare-rule ]

term    := (node, node, node)      // triple pattern
         or (node, node, functor)    // extended triple pattern
         or builtin(node, ... node) // invoke procedural primitive

bhterm  := (node, node, node)      // triple pattern

functor := functorName(node, ... node) // structured literal

node   := uri-ref                // e.g. http://foo.com/eg

```

or prefix:localname	// e.g. rdf:type
or <uri-ref>	// e.g. <myscheme:myuri>
or ?varname	// variable
or 'a literal'	// a plain string literal
or 'lex'^^typeURI	// a typed literal, xsd:/* type names supported
or number	// e.g. 42 or 25.5

If terms are matched by the first part of the rule, the terms following '`->`' are concluded (inferred). Variables are denoted with a '?'. Variables are not typed. A variable will match with any node in the model, which could be requirements or systems (resources in OWL) or relations (object properties in OWL). In our model, requirements are related through object properties. To ensure a variable has a certain type we could have added the following line to our mapping rules:

(?r1 rdf:type Requirement)

However in our case we know that when the following term matches:

(?r1 refines ?r2)

the variables `?r1` and `?r2` must be instances of Requirement. In our rules only requirements can be related through a requires relation. Analogously we assume that if a 'satisfies' is matched in a triple, the left-hand side `?s1` is an instance for set of systems, and the right-hand side `?r1` is a requirement instance:

(?s1 satisfies ?r1)

Therefore, we do not need to check explicitly for the variable's type in the reasoner rules. The following is the rule in Jena that maps the *requires* relation to the set theoretic relations:

```
@include <OWL> .

[requires_to_subclass:

  (?r1 mm:requires ?r2)

  (?s1 inf:satisfies ?r1)

  (?s2 inf:satisfies ?r2) -> (?s1 inf:subClassOf ?s2)]
```

The first line starts with “@include <OWL>.”, which tells the reasoner to import the rules for OWL. This enables the reasoner to reason on OWL and RDF constructs such as transitive object properties. The object properties in the rule’s terms are prefixed with ‘inf:’. This prefix refers to the inference model. The rule states that if $?r1$ requires $?r2$, $?s1$ satisfies $?r1$, and $?s2$ satisfies $?r2$, then we have ($?s1$ is a sub set of $?s2$). The subset relation between the sets of systems are represented by the *subClassOf* construct in OWL. The following Jena rule maps the subset relation to the *requires* relation:

```
[subclass_to_requires:
  (?s1 inf:subClassOf ?s2)
  (?s1 inf:satisfies ?r1)
  (?s2 inf:satisfies ?r2) -> (?r1 mm:requires ?r2)]
```

If there exists a conflicts relation between two requirements, then their sets of systems are disjoint (i.e. there is no system satisfying both requirements). The other way around also holds: if sets of systems are disjoint, the requirements they satisfy are conflicting. Two rules are listed in the following:

```
[conflicts_to_disjoint: (?r1 mm:conflicts ?r2)
  (?s1 inf:satisfies ?r1)
  (?s2 inf:satisfies ?r2) -> (?s1 inf:disjointWith ?s2)
  (?s2 inf:disjointWith ?s1)]
```

```
[disjoint_to_conflicts: (?s1 inf:disjointWith ?s2)
  (?s1 inf:satisfies ?r1)
  (?s2 inf:satisfies ?r2) -> (?r1 mm:conflicts ?r2)]
```

The concluding terms of the first rule (conflicts_to_disjoint) are stating a *disjointWith* relation in both directions because the symmetry of the *disjointWith* property is not handled properly by the JENA reasoner.

We need an additional rule to ensure the so-called ‘permeation of disjointness’, which states that subsets of disjoint sets are also disjoint:

```
[subclass_also_disjoint:
  (?s1 inf:subClassOf ?s2)
  (?s2 inf:disjointWith ?s3) -> (?s1 inf:disjointWith ?s3)]
```

The *partially_refines* relation needs a specific approach since it is a special combination of the *refines* and *contains* relations. Two rules for the *partially_refines* relation are the following:

```
[temp_req_to_p_ref1: (?r1 partially_refines ?r2 )
  <- (?r1 refines ?rt)
      (?r2 contains ?rt)
      (?rt isTemporary 'true'^^xsd:boolean )]

[temp_req_to_p_ref2: (?r1 partially_refines ?r2 )
  <- (?rt contains ?r1)
      (?rt refines ?r2)
      (?rt isTemporary 'true'^^xsd:boolean )]
```

Since we need to distinguish the temporary requirement from the given requirements, we added a data type property named *isTemporary* to the Requirement type. By using the term (*?rt isTemporary 'true'^^xsd:boolean*) we make sure the variable *?rt* is bound to a temporary requirement.

The rules above only match on a temporary requirement. These two rules are of a different type than the other rules. They use '*<-*', and have the concluding terms before the matching terms. This type of rule is called a backward rule, as opposed to the forward rules. Backward rules can be seen as 'goal-driven' rules because they match and execute when the reasoning engine queries to satisfy a certain goal. Forward rules are 'data-driven'. They trigger on given data to infer new triples.

Mapping Requirements Relations to Relations between Formulas. We map the requirements relations *contains*, *refines*, and *partially_refines* to the relations between the formulas. First we define the relations *all-in-part*, *all-in-whole*, *some-implies-in*, *all-implies-in*, *all-equals-in* between formulas. We would like to capture the relations among clauses: implication

and just repetition, and also the coverage among clauses: either all clauses are related or part of them. Let xs and ys be sets of clauses in conjunctive normal form.

(38) $\text{all-in-part}(xs, ys)$

\equiv_{def} For each clause c_{xs} in CNF of xs , there is a distinct clause c_{ys} in CNF of ys where either c_{xs} is equal² to c_{ys} or c_{xs} implies³ c_{ys} . The number of clauses in CNF of xs is smaller than the number of clauses in CNF of ys .

(39) $\text{all-in-whole}(xs, ys)$

\equiv_{def} For each clause c_{xs} in CNF of xs , there is a distinct clause c_{ys} in CNF of ys where either c_{xs} is equal to c_{ys} or c_{xs} implies c_{ys} . The number of clauses in CNF of xs is equal to the number of clauses in CNF of ys .

(40) $\text{some-implies-in}(xs, ys)$

\equiv_{def} There is at least one clause c_{xs} in CNF of xs where c_{xs} implies a clause c_{ys} in CNF of ys and c_{xs} is not equal to c_{ys} .

(41) $\text{all-implies-in}(xs, ys)$

\equiv_{def} For each clause c_{xs} in CNF of xs , there is a distinct clause c_{ys} in CNF of ys where c_{xs} implies c_{ys} and c_{xs} is not equal to c_{ys} .

(42) $\text{all-equals-in}(xs, ys)$

\equiv_{def} For each clause c_{xs} in CNF of xs , there is a distinct clause c_{ys} in CNF of ys where c_{xs} is equal to c_{ys} .

We have the following mappings:

$\text{all-in-whole}(P_1, P_2) \wedge \text{some-implies-in}(P_1, P_2)$ iff R_1 refines R_2

$\text{all-in-part}(P_1, P_2) \wedge \text{all-implies-in}(P_1, P_2)$ iff R_1 partially-refines R_2

$\text{all-in-part}(P_2, P_1) \wedge \text{all-equals-in}(P_2, P_1)$ iff R_1 contains R_2

² If two formulas have the same predicate symbols and arguments, these two formulas are equal.

³ The logical connective *implies* in first-order logic.

The following is the Jena rule that maps the *refines* relation to the *all-in-whole* and *some-implies-in* formula relations:

```
[map_refines_to_formulas:
  (?r1 mm:refines ?r2)
  (?p1 inf:formulas ?r1)
  (?p2 inf:formulas ?r2) -> (?p1 cons:all_in_whole ?p2)
  (?p1 cons:some_implies_in ?p2)]
```

The rule states that if $?r_1$ refines $?r_2$, $?p_1$ is a formula of $?r_1$ and $?p_2$ is a formula of $?r_2$, then we have $(?p_1 \text{ all-in-whole } ?p_2)$ and $(?p_1 \text{ some-implies-in } ?p_2)$. Another rule maps the *all-in-whole* and *some-implies-in* formula relations to the *refines* relation:

```
[map_formulas_to_refines:
  (?p1 cons:all_in_whole ?p2)
  (?p1 cons:some_implies_in ?p2)
  (?p1 inf:formulas ?r1)
  (?p2 inf:formulas ?r2) -> (?r1 mm:refines ?r2)]
```

The rule states that if we have $(?p_1 \text{ all-in-whole } ?p_2)$ and $(?p_1 \text{ some-implies-in } ?p_2)$ where $?p_1$ is a formula of $?r_1$ and $?p_2$ is a formula of $?r_2$, then $?r_1$ refines $?r_2$.

The following is the Jena rule that maps the *partially refines* relation to the *all-in-part* and *all-implements-in* formula relations:

```
[map_part_ref_to_formulas:
  (?r1 mm:partially_refines ?r2)
  (?p1 inf:formulas ?r1)
  (?p2 inf:formulas ?r2) -> (?p1 cons:all_in_part ?p2)
  (?p1 cons:all_implements_in ?p2)]
```

The rule states that if $?r1$ partially refines $?r2$, $?p1$ is a formula of $?r1$ and $?p2$ is a formula of $?r2$, then we have ($?p1$ all-in-part $?p2$) and ($?p1$ all-implies-in $?p2$). Another rule maps the *all-in-part* and *all-implies-in* formula relations to the *partially refines* relation:

[map_formulas_to_part_ref:

```
(?p1 cons:all_in_part ?p2)
(?p1 cons:all_implies_in ?p2)
(?p1 inf:formulas ?r1)
(?p2 inf:formulas ?r2) -> (?r1 mm:partially_refines ?r2)]
```

The rule states that if we have ($?p1$ all-in-part $?p2$) and ($?p1$ all-implies-in $?p2$) where $?p1$ is a formula of $?r1$ and $?p2$ is a formula of $?r2$, then $?r1$ partially refines $?r2$.

The following is the Jena rule that maps the *contains* relation to the *all-in-part* and *all-equals-in* formula relations:

[map_contains_to_formulas:

```
(?r1 mm:contains ?r2)
(?p1 inf:formulas ?r1)
(?p2 inf:formulas ?r2) -> (?p2 cons:all_in_part ?p1)
(?p2 cons:all_equals_in ?p1)]
```

The rule states that if $?r1$ contains $?r2$, $?p1$ is a formula of $?r1$ and $?p2$ is a formula of $?r2$, then we have ($?p2$ all-in-part $?p1$) and ($?p2$ all-equals-in $?p1$). Another rule maps the *all-in-part* and *all-equals-in* formula relations to the *contains* relation:

[map_formulas_to_contains:

```
(?p2 cons:all_in_part ?p1)
(?p2 cons:all_equals_in ?p1)
(?p1 inf:formulas ?r1)
(?p2 inf:formulas ?r2) -> (?r1 mm:contains ?r2)]
```

The rule states that if we have ($?p2$ all-in-part $?p1$) and ($?p2$ all-equals-in $?p1$) where $?p1$ is a formula of $?r1$ and $?p2$ is a formula of $?r2$, then $?r1$ contains $?r2$.

We have the following properties for the formula relations:

all-in-whole, *all-in-part*, *all-implies-in*, and *some-implies-in* relations are transitive

all-in-part and *all-in-whole* relations are disjoint

all>equals-in and *some-implies-in* are disjoint

all>equals-in and *all-implies-in* are disjoint

We have the following inferences for the relations between formulas:

$$(P_1 \text{ all-in-part } P_2) \wedge (P_2 \text{ all-in-whole } P_3) \rightarrow (P_1 \text{ all-in-part } P_3)$$

$$(P_1 \text{ all-in-whole } P_2) \wedge (P_2 \text{ all-in-part } P_3) \rightarrow (P_1 \text{ all-in-part } P_3)$$

$$(P_1 \text{ some-implies-in } P_2) \wedge (P_2 \text{ all-implies-in } P_3) \rightarrow (P_1 \text{ all-implies-in } P_3)$$

$$(P_1 \text{ all-implies-in } P_2) \wedge (P_2 \text{ some-implies-in } P_3) \rightarrow (P_1 \text{ all-implies-in } P_3)$$

$$(P_1 \text{ some-implies-in } P_2) \wedge (P_2 \text{ all>equals-in } P_3) \rightarrow (P_1 \text{ some-implies-in } P_3)$$

$$(P_1 \text{ all-implies-in } P_2) \wedge (P_2 \text{ all>equals-in } P_3) \rightarrow (P_1 \text{ all-implies-in } P_3)$$

$$(P_1 \text{ all>equals-in } P_2) \wedge (P_2 \text{ all-implies-in } P_3) \rightarrow (P_1 \text{ all-implies-in } P_3)$$

We manually check the combinations of the formula relations to derive the inferences given above and to check the completeness of the inferences. The properties and inferences for the formula relations are implemented with the Jena rule language like in the following:

```
[formula_rule_1: (?p1 cons:all_in_part ?p2)
  (?p2 cons:all_in_whole ?p3) -> (?p1 cons:all_in_part ?p3)]
```

The rule states that if ($?p1$ all-in-part $?p2$) and ($?p2$ all-in-whole $?p3$), then we have ($?p1$ all-in-part $?p3$). The rest of the Jena rules for the properties and inferences for the formula relations can be found in Appendix C.

Inferencing. The requirements relations are mapped to facts which concern set and formula relations. The Jena reasoner is capable of automatically inferring new facts based on the properties of the facts and inferences encoded in the Jena rule language. The inferred facts

are mapped back to the requirements relations. For the following inference rule we show how new facts are automatically inferred and mapped back to the relations.

- $(R_1 \text{ partially-refines } R_2) \wedge (R_1 \text{ contains } R_3) \rightarrow (R_3 \text{ partially-refines } R_2)$

$(R_1 \text{ partially refines } R_2)$ is mapped to facts which concern formula relations with the following Jena rule:

[map_part_ref_to_formulas:

```
(?r1 mm:partially_refines ?r2)
(?p1 inf:formulas ?r1)
(?p2 inf:formulas ?r2) -> (?p1 cons:all_in_part ?p2)
(?p1 cons:all_implies_in ?p2)]
```

Since R_1 *partially refines* R_2 , we have $(p1 \text{ all-in-part } p2)$ and $(p1 \text{ all-implies-in } p2)$ where $p1$ is a formula of R_1 and $p2$ is a formula of R_2 . $(R_1 \text{ contains } R_3)$ is mapped to the formula relations with the following Jena rule:

[map_contains_to_formulas:

```
(?r1 mm:contains ?r2)
(?p1 inf:formulas ?r1)
(?p2 inf:formulas ?r2) -> (?p2 cons:all_in_part ?p1)
(?p2 cons:all_equals_in ?p1)]
```

Since R_1 *contains* R_3 , we have $(p3 \text{ all-in-part } p1)$ and $(p3 \text{ all-equals-in } p1)$ where $p1$ is a formula of R_1 and $p3$ is a formula of R_3 . The *all-in-part* relation is transitive. The reasoner takes $(p3 \text{ all-in-part } p1)$ and $(p1 \text{ all-in-part } p2)$, and infers $(p3 \text{ all-in-part } p2)$. The facts $(p3 \text{ all-equals-in } p1)$ and $(p1 \text{ all-implies-in } p2)$ are matched by the following Jena rule:

[formula_rule_7: (?p1 cons:all_equals_in ?p2)

```
(?p2 cons:all_implies_in ?p3) -> (?p1 cons:all_implies_in ?p3)]
```

$(p3 \text{ all-implies-in } p2)$ is inferred. We have $(p3 \text{ all-in-part } p2)$ and $(p3 \text{ all-implies-in } p2)$ as inferred facts. These inferred facts are mapped to the *partially refines* relation by the following rule:

```
[map_formulas_to_part_ref:
  (?p1 cons:all_in_part ?p2)
  (?p1 cons:all_implies_in ?p2)
  (?p1 inf:formulas ?r1)
  (?p2 inf:formulas ?r2) -> (?r1 mm:partially_refines ?r2)]
```

We have (R_3 partially-refines R_2) as inferred.

Consistency Checking. The reasoner is capable of automatically identifying contradicting facts. The following is a proof of one of the consistency checks that uses the formula relations.

Inconsistency: $(R_1 \text{ refines } R_2) \wedge (R_1 \text{ contains } R_2)$

Proof: Let R_1 refines R_2 .

= {By mapping the refines relation to all-in-whole and some-implies-in relations}

(P₁ all-in-whole P₂) \wedge (P₁ some-implies-in P₂) **(a)**

Let R_1 contains R_2 .

= {By mapping the contains relation to part-of and not-imply relations}

(P₂ all-in-part P₁) \wedge (P₂ all-equals-in P₁) **(b)**

The all-in-whole relation in (a) and all-in-part relation in (b) are disjoint. They cannot exist between the same formulas together. The all-equals-in relation is symmetric and it contradicts the some-implies-in relation for the same formulas. Therefore, (R_1 refines R_2) and (R_1 contains R_2) contradict one another.

The Jena rules for consistency checking can be found in Appendix D. We manually check the combinations of the formula relations to derive the consistency checking rules and to check the completeness of the rules. The following is one of the Jena rules for consistency checking:

```
[all_in_whole__contradicts__all_in_part:
```

```
  (?p1 cons:all_in_whole ?p2)
```

```
(?p2 cons:all_in_part ?p1) -> addInconsistency('all-in-whole_all_in_part', ?p1, ?p2)]
```

The rule states that if ($?p1$ all-in-whole $?p2$) and ($?p2$ all-in-part $?p1$), then there is an inconsistency.

4.6 Tool Support

We built a tool named TRIC (Tool for Requirements Inferencing and Consistency checking) for automatic inferencing and consistency checking [254]. TRIC and an example requirements model can be downloaded from [245]. In this section, we give the details of the tool. In Section 4.6.1, we describe the usage of the tool in the context of a modeling process. Section 4.6.2 gives the architecture of the tool. Section 4.6.3 describes the main features of the tool with some screenshots.

4.6.1 The Modeling Process

We depict the usage of the tool in a modeling process with inferencing and consistency checking. This process is based on the analysis of activities during modeling of requirements and their relations. Figure 4.3 gives a UML activity diagram of the process.

The process consists of the following activities:

Modeling. This activity takes the requirements document as input and produces the requirements model which is an instance of the requirements metamodel. The requirements model contains requirements and their relations. The definitions given in Section 4.3 are used to specify the requirements relations.

Inferencing and Consistency Checking. The modeling process is forked into two activities: *consistency checking* and *inferencing*. These two activities are processed in parallel. The requirements model is updated with inferred relations. Inconsistent parts of the model are determined, if there are any. Inferencing and consistency checking enrich the set of requirements relations in the requirements model. These two activities are combined because the consistency checking uses the machinery for inferencing and also checks the inconsistencies among inferred relations as well as among given relations.

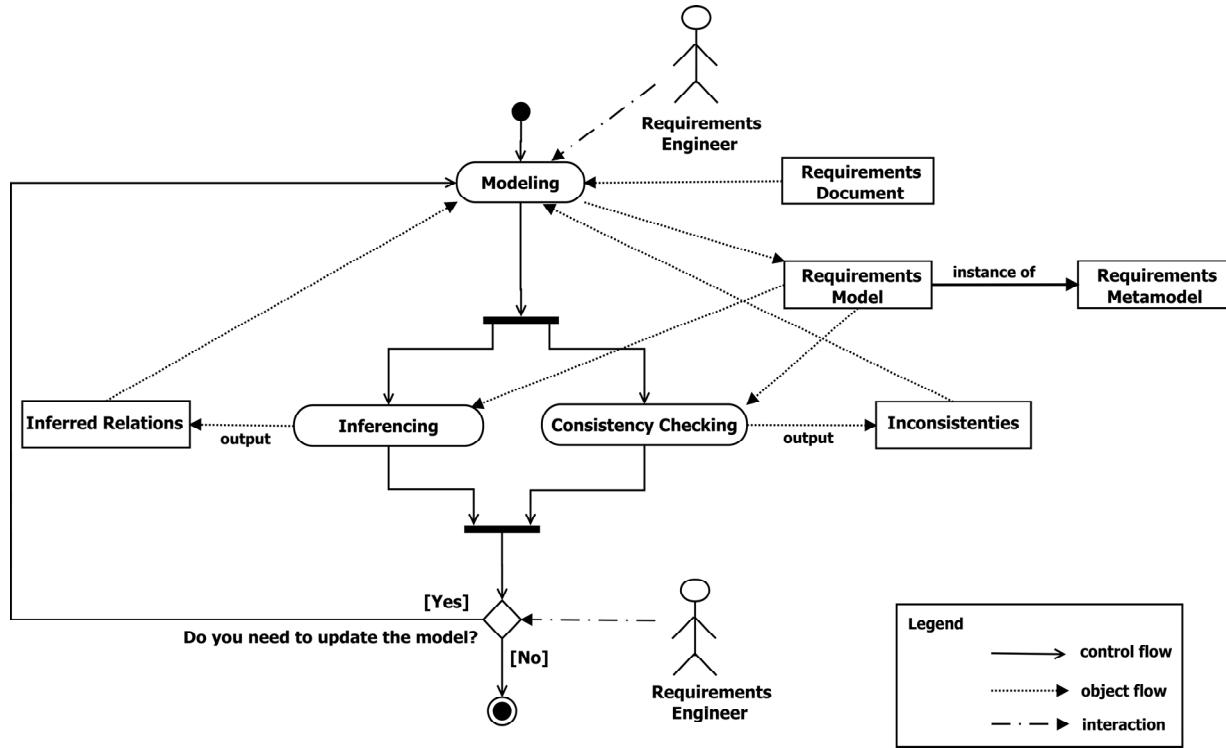


Figure 4.3 Modeling Process with Consistency Checking and Inferencing

Iterating. The process given in Figure 4.3 is iterative: the requirements engineer may return to the modeling activity in order to fix inconsistencies and/or input new requirements and relations. If there is no need to update the model, the process is terminated.

4.6.2 Tool Architecture

The tool architecture is composed of three layers (see Figure 4.4): a) the *User Interface (UI) layer*, b) the *Application Layer*, and c) the *Data Layer*.

User Interface (UI) Layer. This layer supports the modeling activity. The user interface is implemented by using the Eclipse Rich Client Platform (RCP) [218]. The output of the consistency checking and inferencing is represented in a table form. The JGraph library [131] is used for the graphical visualization of this output. The layer provides the following:

- A form-based editor to enter and modify requirements
- An editor to enter and modify relations between requirements
- A matrix view of requirements in the model
- The control of the services provided by the application layer

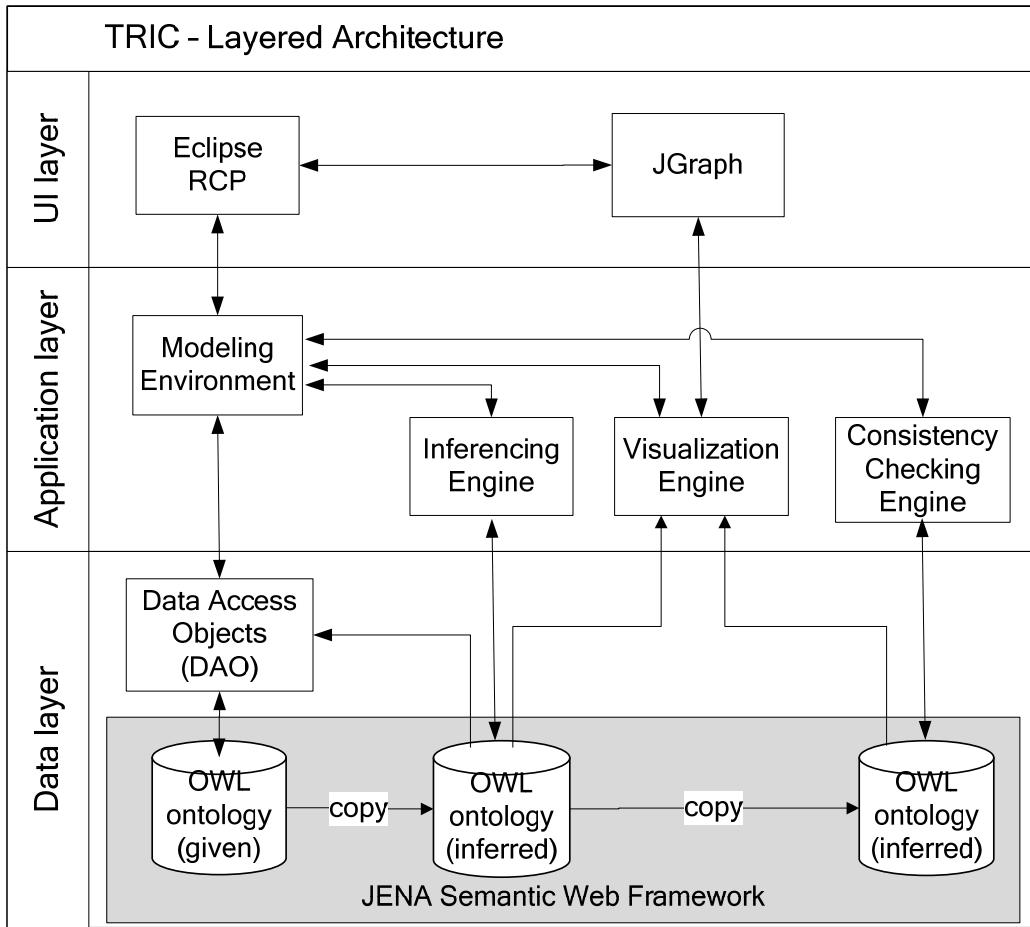


Figure 4.4 Layered Architecture of the Tool

Application Layer. This layer performs the main activities given in Figure 4.3: *consistency checking* and *inferencing*. It contains the components *Modeling Environment*, *Inferencing Engine*, *Consistency Checking Engine*, and *Visualization Engine*. The components provide the following functionalities:

- *Modeling Environment*: allows the creation, storage, and retrieval of requirements models, and bridging the User Interface layer with the Data layer.
- *Inferencing Engine*: infers all implicit relations between requirements, and keeps track of given and inferred relations.
- *Consistency Checking Engine*: allows checking consistency of relations.

- *Visualization Engine*: accesses the Data layer in order to get requirements and relations to be visualized in diagrams. The visualization is done by JGraph in the User Interface layer.

The *Inferencing Engine* component also implements the mappings between requirements relations and their definitions in the formalization. These mappings are required to implement consistency checking and inferencing.

Data Layer. The entered requirements and their relations are stored as an OWL ontology [62] which consists of the requirements metamodel and its instance model in the same file. Therefore, we can use the existing reasoners developed for the semantic web environment. Our formalization is directly mapped to the language features of OWL like transitivity and symmetry of properties. Reasoning on requirements models is done on OWL ontologies. We used Jena [130], a programmatic environment for processing OWL data, with a rule-based inference engine. The engine performs consistency checking and inferencing. One of the advantages of Jena is that it provides derivation trace analysis. The analysis is used in one of the main facilities of the tool: *explaining results of inferencing and consistency checking*. We reason on copies of the given ontology in order to prevent the pollution of the given requirements ontology with inferred relations and inconsistencies. The Data Access Objects (DAO) component is responsible for reading and manipulating models without any dependency on data format.

4.6.3 Tool Features

We describe the most important features of the tool: *managing requirements (add, update, delete requirements and relations)*, *displaying inconsistencies & inferred relations*, and *explaining the results of reasoning*.

Managing requirements. We can add new requirements and update or delete existing requirements. Figure 4.5 gives the GUI for managing requirements which supports the modeling activity in Figure 4.3.

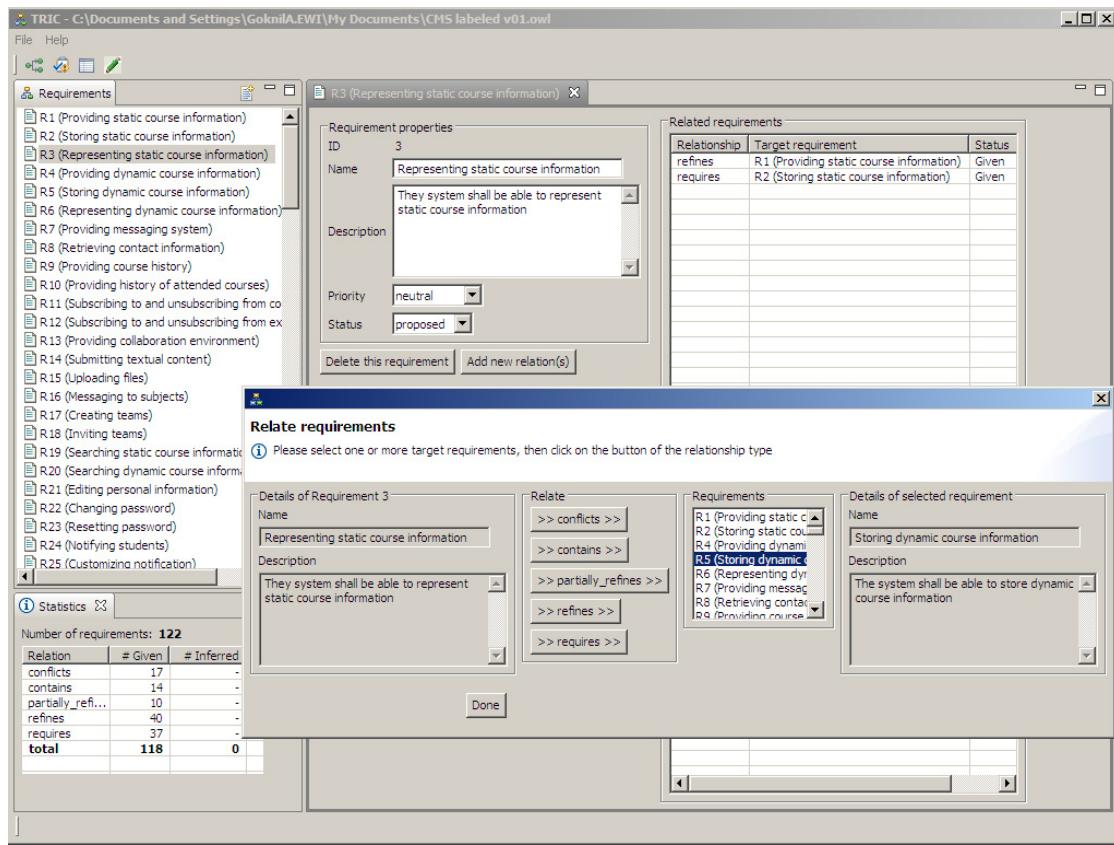


Figure 4.5 GUI for Managing Requirements and Relations

The left-hand side of the window lists the entered requirements. The right-hand side of the window shows details of the selected requirement (R3). The tool gives a warning if a deleted given relation is inferred by the inferencing engine. The *Relate requirements* window opened by the *Add new relation(s)* button is used to select related requirements and relation types.

The tool provides a matrix view in order to represent and manage requirements and relations. Such a view is also available in commercial requirements management tools, such as RequisitePro. Figure 4.6 illustrates the matrix view feature of our tool.

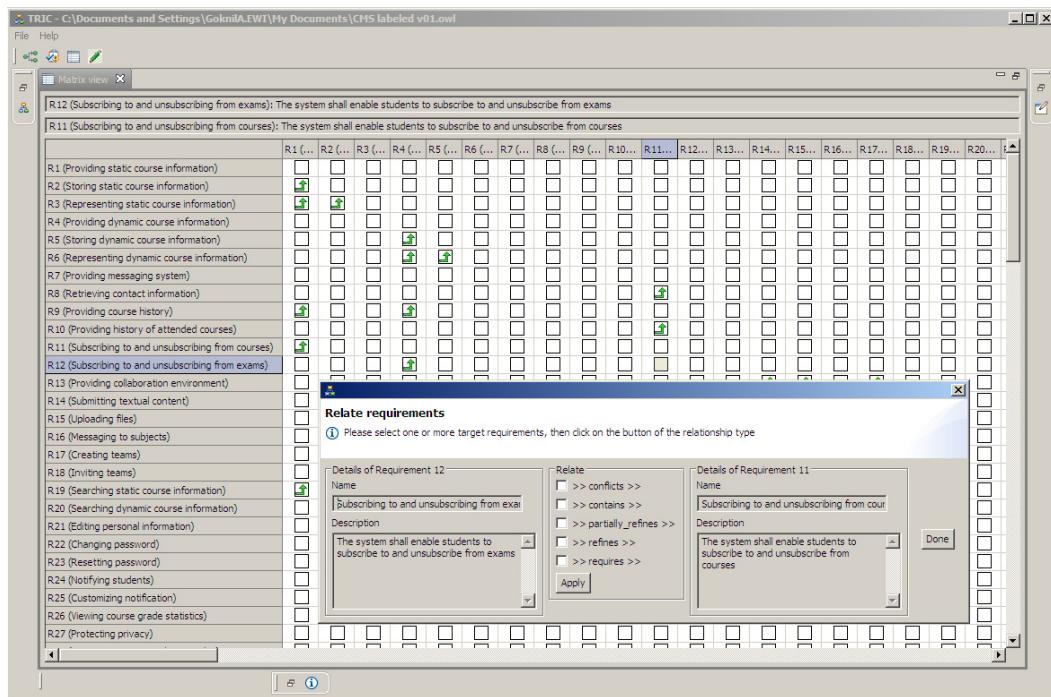


Figure 4.6 Matrix View for Managing Requirements and Relations

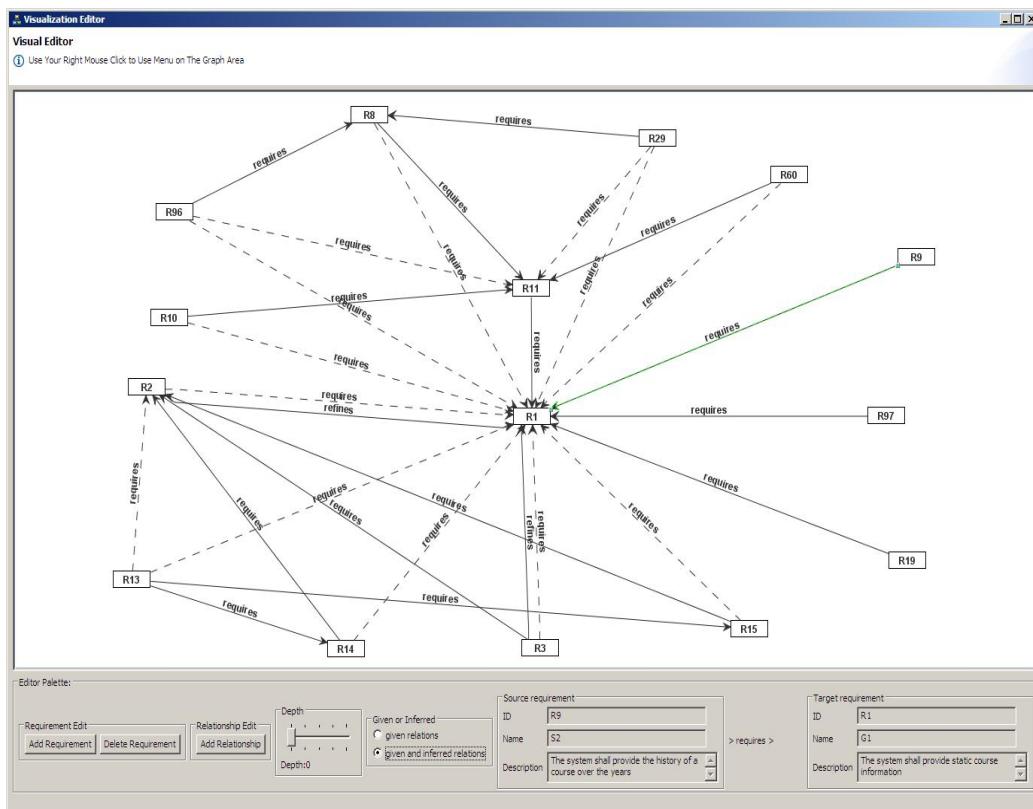


Figure 4.7 Visual Editor for Managing Requirements and Relations

The arrows with direction in the cells denote the existence of requirements relations with their directions. Since there might be multiple relations between two requirements, the tool provides the *Relate requirements* window, which is similar to the window in Figure 4.5.

The matrix view is less usable for large models. We provide a visual editor (see Figure 4.7) in order to improve the usability of the tool for large models. The requirements engineer can select a smaller set of requirements to be shown in a graph.

Displaying inconsistencies and inferred relations. Figure 4.8 gives the screenshot of the tool for output of the inferencing activity.

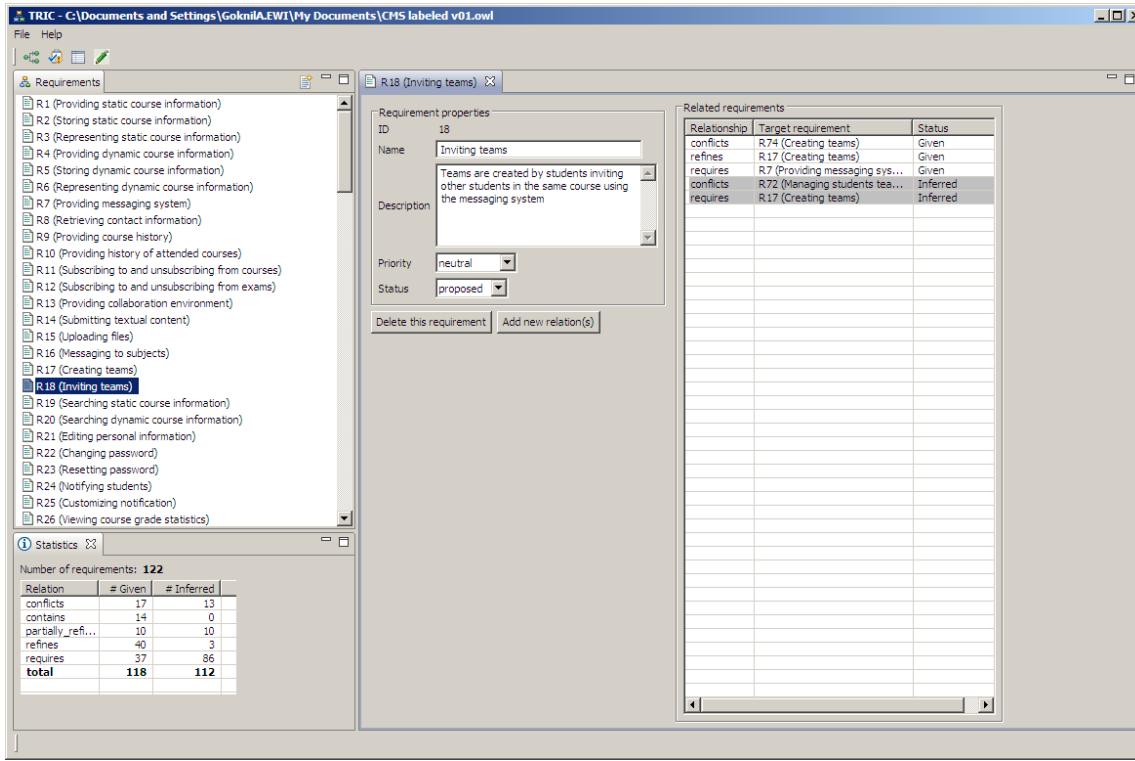


Figure 4.8 Output of the Inferencing Activity

The highlighted relations (*conflicts* and *requires*) in the right part of the window are the inferred relations for the requirement R18. The tool detects contradictions in the model. Figure 4.9 gives the screenshot of output of the consistency checking activity.

Description	Contradicting relations
Both conflicts and requires	[R8 conflicts R98 (inferred), R8 requires R98 (inferred)]
Requirement conflicts with itself	[R60 conflicts R60 (inferred)]
Both conflicts and requires	[R8 requires R48 (inferred), R8 conflicts R48 (inferred)]
Both conflicts and requires	[R8 requires R97 (inferred), R8 conflicts R97 (inferred)]
Requirement conflicts with itself	[R8 conflicts R8 (inferred)]
Both conflicts and requires	[R16 conflicts R98 (inferred), R16 requires R98 (inferred)]
Both conflicts and requires	[R16 requires R48 (inferred), R16 conflicts R48 (inferred)]
Both conflicts and requires	[R60 conflicts R11 (inferred), R60 requires R11 (given)]
Requirement conflicts with itself	[R16 conflicts R16 (inferred)]
Both conflicts and requires	[R16 conflicts R97 (inferred), R16 requires R97 (inferred)]
Both conflicts and partial-refines	[R60 partially_refines R59 (given), R60 conflicts R59 (inferred)]
Both conflicts and requires	[R10 requires R11 (given), R10 conflicts R11 (inferred)]
Both conflicts and requires	[R29 conflicts R48 (inferred), R29 requires R48 (inferred)]
Both conflicts and requires	[R29 requires R11 (inferred), R29 conflicts R11 (inferred)]
Both conflicts and requires	[R16 requires R8 (given), R16 conflicts R8 (inferred)]
Both conflicts and requires	[R10 conflicts R48 (inferred), R10 requires R48 (inferred)]
Requirement conflicts with itself	[R29 conflicts R29 (inferred)]
Both conflicts and requires	[R16 conflicts R11 (inferred), R16 requires R11 (inferred)]
Both conflicts and requires	[R11 conflicts R48 (inferred), R11 requires R48 (given)]
Both conflicts and requires	[R8 requires R11 (given), R8 conflicts R11 (inferred)]
Both conflicts and requires	[R60 requires R98 (inferred), R60 conflicts R98 (inferred)]
Both conflicts and requires	[R60 conflicts R97 (inferred), R60 requires R97 (inferred)]
Requirement conflicts with itself	[R10 conflicts R10 (inferred)]
Both conflicts and requires	[R10 conflicts R97 (inferred), R10 requires R97 (inferred)]
Requirement conflicts with itself	[R11 conflicts R11 (inferred)]
Both conflicts and requires	[R29 conflicts R8 (inferred), R29 requires R8 (given)]
Both conflicts and requires	[R29 conflicts R98 (inferred), R29 requires R98 (inferred)]
Both conflicts and requires	[R10 requires R98 (inferred), R10 conflicts R98 (inferred)]
Both conflicts and requires	[R11 requires R97 (given), R11 conflicts R97 (inferred)]
Both conflicts and requires	[R60 requires R48 (inferred), R60 conflicts R48 (inferred)]
Both conflicts and requires	[R29 requires R97 (inferred), R29 conflicts R97 (inferred)]
Both conflicts and requires	[R11 requires R98 (inferred), R11 conflicts R98 (inferred)]

Figure 4.9 Output of the Consistency Checking Activity

The left part of the window gives descriptions of the inconsistencies; the right part gives the contradicting relations.

Explaining results of reasoning. The requirements engineer may need further explanation of the result from the reasoning in order to update the requirements model. The tool visualizes how inferred relations are derived (see Figure 4.10).

In Figure 4.10, the derivation of the *conflicts* relation between requirements R8 and R59 (dashed arrow) is visualized. Note that this conflicts relation is not an inconsistency itself. The solid arrows indicate the given relations used in the derivation.

The tool also provides an explanation of contradicting relations, for example the inconsistency for requirements R11 and R48 (see Figure 4.11).

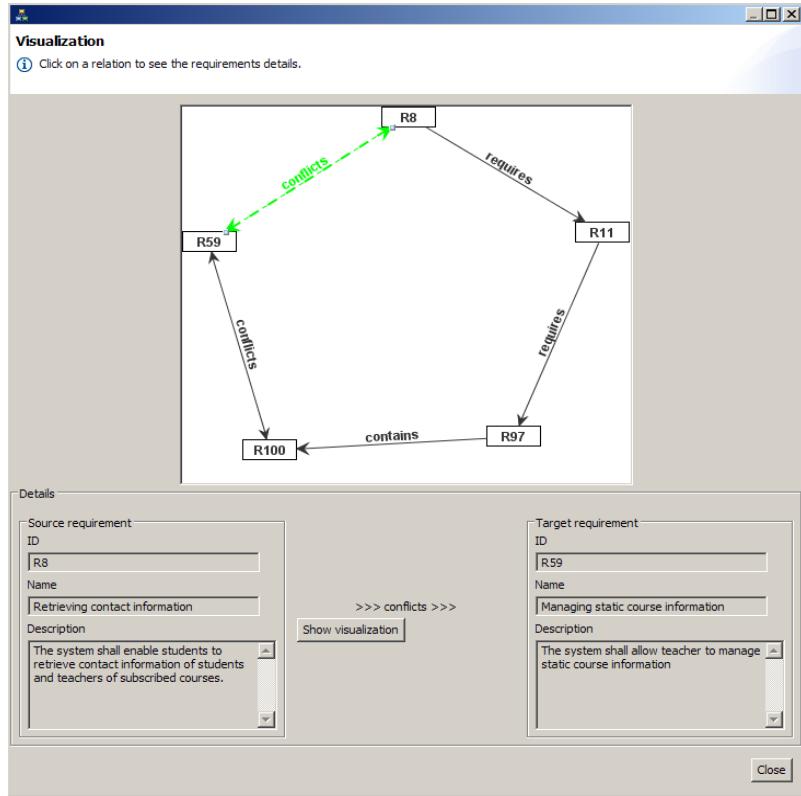


Figure 4.10 Explanation of the Inferred Conflicts Relation between R8 and R59

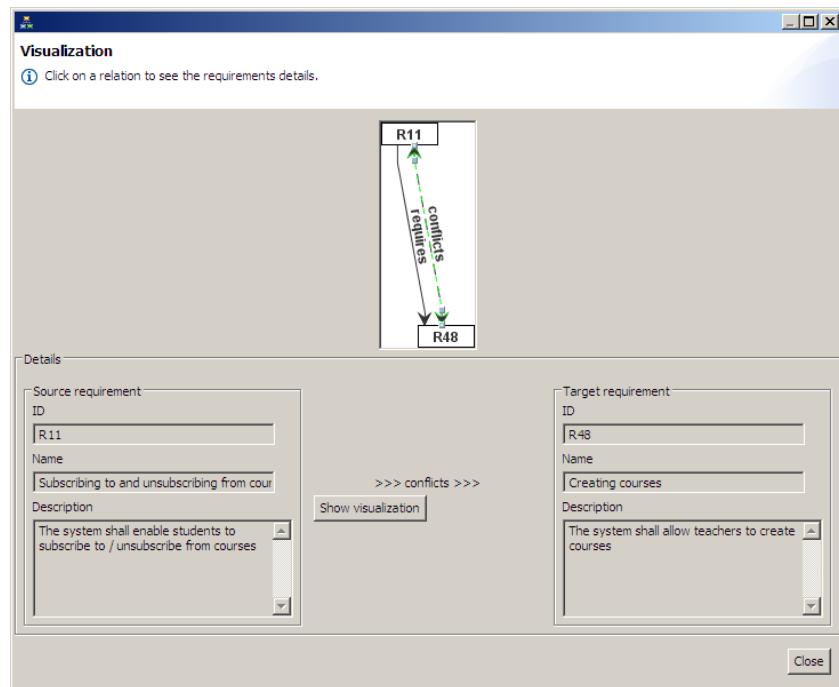


Figure 4.11 Explanation of the Inconsistency for R11 and R48

The solid arrow (the *requires* relation) indicates the given relation in the inconsistency; the dashed arrow (the *conflicts* relation) denotes the inferred relation in the inconsistency. The *Show visualization* button is used to visualize the derivation of the inferred conflicts relation (see Figure 4.12). Since the set of contradicting relations like in Figure 4.11 may contain inferred relations, the visualization in Figure 4.12 helps the requirements engineer to resolve contradictions by identifying all given relations causing the inconsistency.

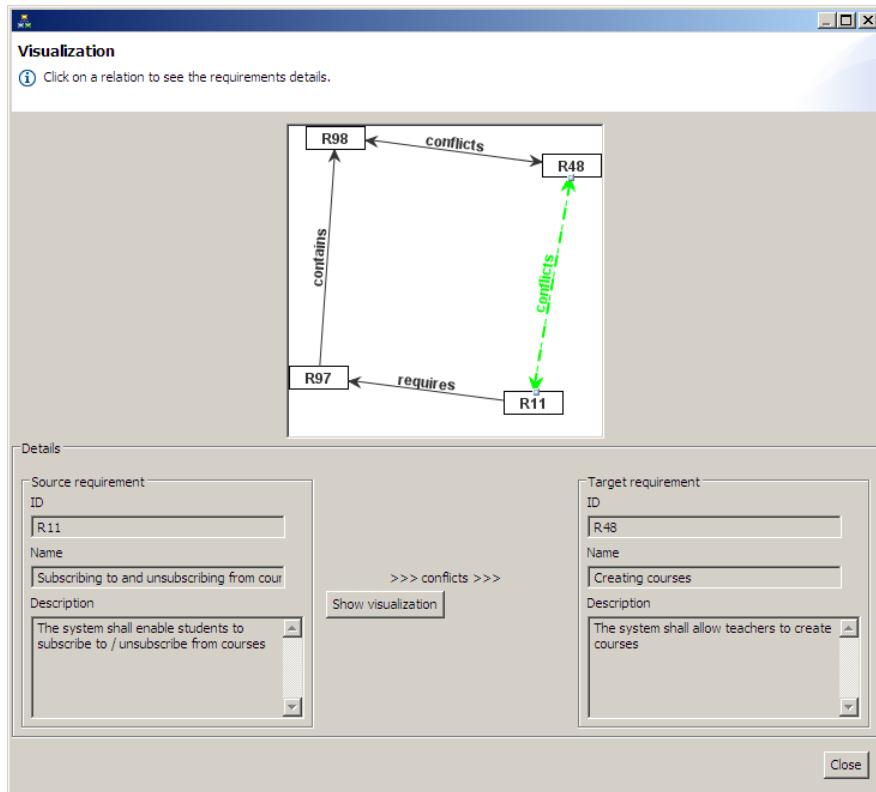


Figure 4.12 Explanation of the Inferred Conflicts Relation in the Inconsistency

Another visualization option provided by the tool is to visualize the requirements related to a selected requirement at a given depth. *Depth* is the maximum number of relations between the requirement and its related requirements in the shortest path. Figure 4.13 shows the requirements related to the requirement R5 at depth 2.

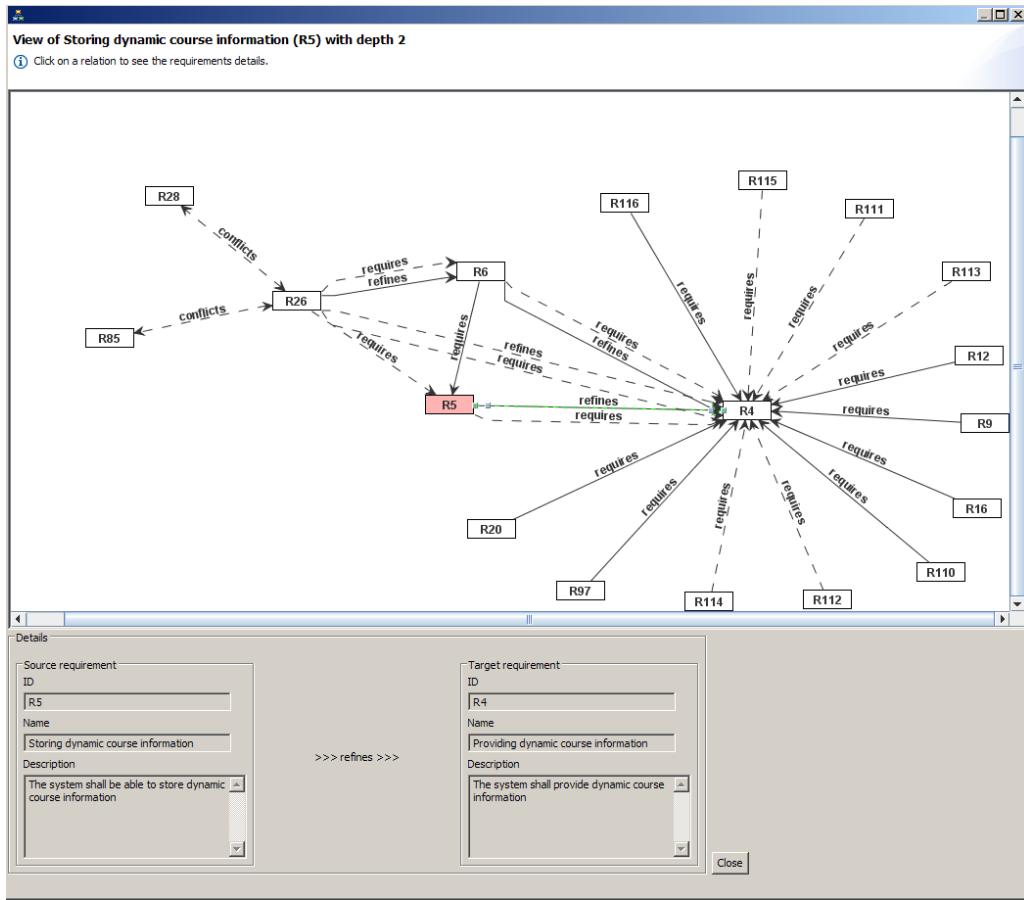


Figure 4.13 Visualization of the Related Requirements for R5 with Depth 2

This visualization option allows showing only a part of the requirements model. It is useful for large models where the matrix view does not scale well.

4.7 Example: Course Management System

In this section, we illustrate our approach and tool support with the CMS example. The CMS requirements document was prepared as a result of a discussion by QuadREAD project members who took the role of stakeholders. No particular inconsistencies and conflicts were inserted intentionally. We aimed at detecting inconsistencies and conflicts as a result of the modeling process. All requirements used in this chapter can be found in Appendix B. We performed two iterations of the modeling process for the example.

- In the first iteration, we modeled the textual requirements and their relations according to the semantics of relation types. We analyzed given and inferred relations and inconsistencies by using the outputs of the tool. The requirements engineer

identifies which relations are valid or invalid based on his/her knowledge of the application domain and the semantics of the relations. He or she decides how to correct invalid given relations by using the feature for explaining the output of reasoning.

- In the second iteration, we updated the model in order to correct the invalid relations. The validity of relations in the model was checked according to the semantics of the relation types. This checking is dependent on the requirements engineer's interpretation of the semantics of the relations.

It should be noted that the conclusions from the example cannot be generalized for our approach, since we still need to apply the approach to a number of industrial and academic case studies with empirical results. The example illustrates potential benefits and limitations of the approach for larger case studies. Section 4.7.1 presents the overall results of the two iterations. Section 4.7.2 gives some inferred relations in the example. In Section 4.7.3, we show some inconsistencies detected in the example requirements model.

4.7.1 Modeling the Requirements

The requirements in the document are grouped by their stakeholders, which are *Student*, *Lecturer*, *System Maintainer*, and *Administration*. The functional and non-functional requirements are separated in the requirements document. There are 122 requirements (94 functional and 28 non-functional requirements). In the document, relations between requirements are not stated explicitly.

In the first iteration, we modeled the document according to our relation types by interpreting the requirements in the document. The execution of the inference engine inferred new relations based on the given relations. As a second step, we run the consistency checker for the requirements model.

The tool reported 32 inconsistent parts in the requirements model. An inconsistent part is a set of relations whose existence causes a contradiction. For example, the *conflicts* and *requires* relations between R29 and R97 cause a contradiction. The output for one of the inconsistent parts is given below:

Inconsistency

Description: “Both conflicts and requires relations”

Contradicting relations:

R29 requires R97 (inferred relation)

R29 conflicts R97 (inferred relation)

In the second iteration, we used the tool feature for explaining the results of reasoning. The feature provides derivation trace analysis of inconsistent parts of the model. Based on this information, we discovered that there are five invalid given *requires* relations, one *refines* relation is actually a *contains* relation, and one *contains* relation is actually a *partially refines* relation in the example. Deleting and updating these relations results in a consistent requirements model. The number of inferred relations is reduced. Table 4.1 gives the number of given and inferred relations, and the number of inconsistencies in the first and second iteration for the CMS example.

Table 4.1 Number of Relations and Inconsistencies in the Example

		Number of Relations per Relation Type						Number of Inconsistencies
First Iteration	Given	Refines	Partially Refines	Requires	Contains	Conflicts	Total	
	Inferred	3	10	122	0	103	238	
Second Iteration	Given	40	10	37	14	17	118	0
	Inferred	3	10	86	0	13	112	

In the first iteration there are 225 *conflicts* and *requires* relations of 238 inferred relations. Updating the model in the second iteration in order to fix the inconsistencies eliminates the inferred invalid *conflicts* and *requires* relations.

In the second iteration, reasoning on the requirements model resulted in 112 inferred relations from 118 given relations. There are 86 *requires* relations of 112 inferred relations. From the formalization of relation types, we know that the *contains* and *refines* relations imply the *requires* relation in the requirements model. Therefore, we were expecting that the number of inferred *requires* relations would be more than the total number of given *contains* and *refines* relations. Fifty-four of these 86 inferred *requires* relations are inferred from the given *contains* and *refines* relations. Other *requires* relations are inferred by using the transitive property of the *requires* relation and the combination of the *requires* relation with *contains* and *refines* relations.

As a result of reasoning, we have 13 inferred *conflicts* relations from 17 given *conflicts* relations. All these *conflicts* relations are inferred because of the combination of the *conflicts* relation with the *requires* and *contains* relations.

In the requirements document, the containment hierarchy has only one level. Since the transitive property of the *contains* relation is the only way to infer the *contains* relation according to its semantics, the tool does not infer any new *contains* relations. We have only three inferred *refines* relations from 40 given *refines* relations by using the transitive property of the *refines* relation. On the other hand, 10 *partially refines* relations are inferred from 10 given *partially refines* relations.

4.7.2 Inferring Requirements Relations

In this section, we describe some inferred relations in the example. The example in Figure 4.14 illustrates the inferencing for the following requirements:

R5: The system shall be able to store *dynamic course information*.

R6: The system shall be able to represent *dynamic course information*.

R26: The system shall allow students to view course grade statistics per semester.

In the glossary of the requirements document (see Appendix B), dynamic course information is expressed as information (news messages, archived files, and roster) about a course which changes while a course is given. In the requirements model, the following relations are given: (R26 refines R6) and (R6 requires R5). When we run our tool over the requirements model, the relation (R26 requires R5) is inferred (dashed line in Figure 4.14).

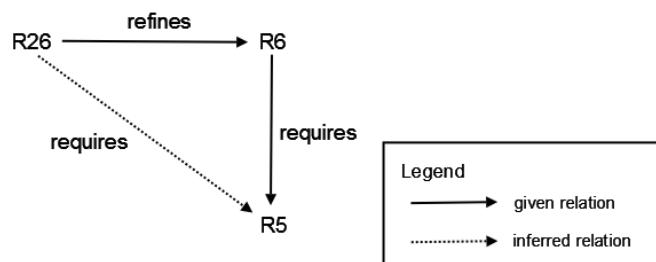


Figure 4.14 Example with Inferred Requires Relation

Grade statistics are dynamic course information. The system needs to store dynamic course information in order to allow students to view course grade statistics per semester. Therefore, we confirm that the inferred relation (R26 requires R5) is a valid relation in the model.

The interpretation of requirements depends on the requirements engineer. In the example, we discovered some invalid given relations. The tool feature for explaining the inferencing results supports our analysis of (in)valid given relations based on inferred relations. Figure 4.15 depicts the analysis of one inferred relation to identify invalid given relations.

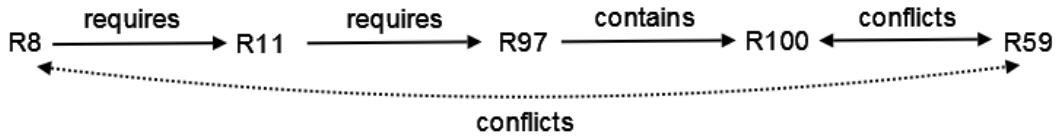


Figure 4.15 Analysis of the Inferred Relation to Identify Invalid Given Relations

Although there is an inferred conflicts relation between requirements R8 and R59, these two requirements are not in conflict. These requirements are the following:

R8: The system shall enable students to retrieve contact information of students and lecturers of subscribed courses.

R59: The system shall allow lecturers to manage *static course information*.

When we analyzed the given relations used to infer conflicts relations in Figure 4.15, we concluded that the given relation (R11 requires R97) is not a valid relation. These two requirements are the following:

R11: The system shall enable students to subscribe to and unsubscribe from courses.

R97: The system shall allow only the administration to *manage* courses.

R11 does not require R97 to be fulfilled. The invalid input causes the invalid output of the inferencing. The tool helps to identify candidate invalid given relations in the example.

4.7.3 Checking Consistency

In the previous section we treated conflicts relations, which are modeled by the requirements engineer. Here, we discuss inconsistencies, that is, contradictions among relations which are detected by our tool. We will depict how we fix an inconsistent part by using the output of our tool. The example in Figure 4.16 illustrates this part. The requirements are:

R11: The system shall enable students to subscribe to and unsubscribe from courses.

R48: The system shall allow lecturers to create courses.

The consistency checking engine reports that conflicts and requires relations between R11 and R48 cause a contradiction. The relation (R11 requires R48) is a given relation. When we

re-analyzed requirements R11 and R48, we concluded that this requires relation is an invalid relation. Since there might be hard coded courses in the system, the students can subscribe to and unsubscribe from these courses without any need to create courses.

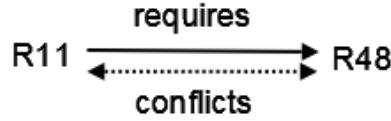


Figure 4.16 Inconsistent Part in the Example Model

Since the relation (R11 conflicts R48) is an inferred relation, we need derivation trace analysis for this relation. Figure 4.17 gives the analysis of the inferred relation in the inconsistent part of the model.

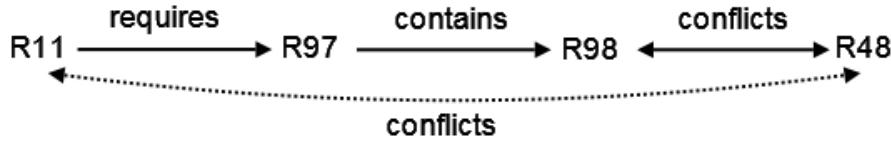


Figure 4.17 Analysis of the Inferred Relation in the Inconsistent Part of the Model

When we checked the given relations in Figure 4.17, we found that the given relation (R11 requires R97) is an invalid relation, modeled incorrectly in the first iteration. This is the same relation we identify in the analysis of the inferred relation in Figure 4.15. We removed the *requires* relation between R11 and R97 to fix the inconsistent part in Figure 4.16. This example illustrates how the tool helps localizing invalid relations.

4.8 Related Work

We classify the related work in four categories: *Requirements Relations*, *Requirements Metamodeling*, *Requirements Reasoning*, and *Tool Support*.

4.8.1 Requirements Relations

We studied literature about requirements relation types and their semantics. Dahlstedt and Persson [59] address requirements relations (they call a relation an “interdependency”) from a traceability perspective. They give an overview of requirements relations research and present a model of fundamental relation types. There is a classification (*structural*, *constrain*, and *cost/value interdependencies*) of fundamental interdependency types which includes some of the relations (*refines*, *requires*, and *conflicts*) we also use in our approach. The need to understand the nature of requirements relations and their influence on software

development activities such as change management are stated. However, there is no formal semantics for the relations. Carlshamre et al. [43] run an industrial survey of requirements in software product release planning. Their aim is to learn about the nature of interdependencies in general, to be able to classify them, and to assess the relative frequency of different classes. The results show that roughly 20% of the requirements are responsible for 75% of the interdependencies and only a few requirements are singular. It is expected to find conflicting requirements in the survey, since this relation is common in the literature. However, no such dependencies are identified. Apparently conflicts had already been eliminated in the documents.

Although the two studies mentioned above motivate the need for requirements relations, no much attention is paid for how to give formal semantics of the relations. Aizenbud-Reshef et al. [6] state the need for semantics of traceability links in general. They present an approach to defining operational semantics for traceability in UML which captures more precisely the intended meaning of various types of traceability. The main goal is achieving automated consistency management of UML models. The semantic property of a traceability relationship is a triplet (*event*, *condition*, and *actions*). This triplet is very much dependent on change impact analysis. For instance, an event indicates a change in a model. Conditions help to differentiate between events. Actions describe what should and should not be done when a specific event has occurred. Therefore, it is hard to use the semantics in [6] for other purposes like inferencing and consistency checking of trace relations. On the other hand, the semantics formalized with FOL in this chapter can be considered as more general and suitable for different purposes. In [96] and [95], we use our semantics for inferencing, consistency checking, and change impact analysis in requirements models.

Lee et al. [152] studies relationships between soft functional requirements based on fuzzy logic. The types of relations between soft functional requirements are classified as *conflicting*, *irrelevant*, *cooperative*, *counterbalance* and *independent*. These relation types are formalized by using fuzzy logic. Contrary to our approach, the relation types in [152] are specialized for imprecise requirements and they are used for trade-off analysis.

The survey in [222] introduces Requirements Interaction Management (RIM), which is concerned with the analysis and management of dependencies among the requirements. One of the activities in RIM, is reasoning on requirements interactions. Conflict detection methods for reasoning are introduced in five categories: *domain model*, *theorem model*, *scenario analysis*, *modeling checking* and *executing monitoring* methods. We consider our work in the scope of the domain model method. The domain model method is summarized in the survey that a domain model of system requirements interactions is used to identify interactions at the

requirement level. We consider that our requirements metamodel is our domain model of requirements relations which stand for requirements interactions to identify relations between requirements.

4.8.2 Requirements Metamodeling

A number of approaches in MDE address modeling requirements and their relations from a traceability perspective. Vicente-Chicote et al. [255] describe a requirements metamodel and a modeling environment. The environment supports: graphical requirements models, their validation against the metamodel and against a set of constraints written in Object Constraint Language (OCL), and automatic generation of a navigable Software Requirements Specification document (SRS). In the requirements metamodel, there are three types of trace links between requirements: *DependenceTrace*, *InfluenceTrace*, and *ParentChildTrace*. The relations are defined informally.

Baudry et al. [17] introduce a metamodel for requirements and present how they use it on top of a constrained natural language for requirements definitions. The requirements metamodel captures functional requirements as use cases with pre-conditions and post-conditions that constrain the activation of use cases. Operations are added in the metamodel in order to simulate requirements models. The goal of the simulation is to instantiate the use cases, replacing the formal parameters with actual values defined in an initial configuration. The metamodel does not capture the static part of requirements. It does not have the notion of requirements relations. On the other hand, our approach covers the static aspects of requirements including non-functional requirements and reasoning on requirements relations. In [37], a model-driven mechanism is proposed to merge different requirement specifications and reveal inconsistencies between them by using a requirements metamodel. The requirements metamodel is mainly used to produce a requirements model from a given requirements document. Requirements relations are not typed and lack semantics. Consistency checking and inferencing for requirements relations are not supported.

Some authors [111] [236] use the UML profiling mechanism in a goal-oriented requirements engineering approach. Heaven et al. [111] introduce a profile that allows the KAOS model [250] to be represented in UML. They also provide an integration of requirements models with lower level design models. Supakkul et al. [236] use the UML profiling mechanism to provide an integrated modeling language for functional and non-functional requirements that are mostly specified by using different notations. These two works aim at a metamodel for goal-oriented requirements engineering rather than reasoning over requirements.

SysML [200] [231] uses the UML profiling mechanism to provide modeling constructs that represent text-based requirements and relate them to other modeling elements. The relation

types for requirements in SysML are *derive*, *copy*, and *contain*. SysML also provides a stereotype mechanism that allows the requirements engineer to specify their own relation types. The main goal of SysML requirements diagrams is to represent the requirements and their relations. Formal semantics of relation types is not considered. The definitions of the relations tend to be ambiguous. No reasoning facility for requirements is provided.

Vogel and Mantell [256] provides a UML profile that allows the modeling of stakeholders, requirements and test cases. The profile has two parts: *Stakeholders* and *Requirements*. The first part includes entities for types of stakeholders such as *User*, *Project Stakeholder*, *Supplier* and *Customer*. The second part of the profile contains entities for *TestCase* and types of requirements such as *Performance Requirement* and *Functional Requirement*. The profile contains entities similar to entities in our requirements metamodel. However, there is no requirements relation in [256].

COMET [55], a requirements modeling method, provides a requirements metamodel which is an extension to the use case concept of UML. COMET considers the UML use cases as the only requirements specification method. The requirements metamodel includes a use case entity with interacting roles, scenario which is the detailed description of the use case, goal entity, and the requirement entity represented by the use case. Requirements relations are not represented in the requirements metamodel of COMET.

Navarro et al. [189] propose a customization approach for requirements metamodels. They propose a core requirements metamodel which is generic and considers only *Artifact* and *Dependency* as core entities. The metamodel does not contain concrete types for requirements relations. This disallows the application of inference rules for the core relations to customized entities. The Requirements Interchange Format (RIF) [220] structures requirements and their attributes, types, access permissions, and relationships. It is defined as an XML schema. Its data model has generic entities and relations like *Information Type*, *Association*, and *Generalization*. These entities can be formalized to reason about requirements and their relations. Ramesh et al. [215] propose reference models for requirements traceability. The models include basic entities like *Stakeholder*, *Object*, and *Source*. Relations between different software artifacts and requirements are captured.

Some papers address domain-specific requirements models. Koch et al. [143] propose a requirements metamodel specialized for Web systems. They identify the general structure of Web systems in order to define the requirements metamodel. The requirements metamodel for web requirements, presented by Escalona and Aragon [75], is divided into two packages: the *Behavior* and the *Structure*. In the behavior package, concepts such as *WebActor* and *WebUseCase* related to the behavior of the system presented. In the structure package, any

information storage for the system is represented. Molina et al. [180] [181] propose another web engineering requirements metamodel as an extension that can be integrated with existing web engineering methodologies. A tool is provided as an eclipse plug-in that accompanies the metamodel presented in [180] [181]. The metamodel is extended with general security concepts in [226] in order to define a domain specific language for security requirements. In [178], Molina presents a measurable requirements metamodel which extends the requirements metamodel in [180] [181]. The measurable requirements metamodel supports the elicitation of measurable requirements based on the explicit connection of goals, requirements, and measures. Moon et al. [183] propose a methodology for producing requirements that can be considered as a core asset in the product line. Ceron et al. [44] discuss requirements modeling in the context of product lines. They propose a metamodel for requirements that contains both the common and variable parts. Lopez et al. [164] propose a metamodel for requirements reuse as a conceptual schema to integrate semiformal requirement diagrams into a reuse strategy. The requirements metamodel is used to integrate different abstraction levels for requirements definitions. All these domain-specific approaches aim at providing a structure for representing requirements and their relations. Some of them do not contain types of requirements relations and most of them only provide informal definitions of their relations.

In [179] there is a review of requirements metamodels in literature. Loniewski et al. [162] presents a review of the use of requirements engineering techniques in Model-Driven Engineering. They do not focus on requirements metamodels but MDE approaches that use requirements metamodels are summarized and reviewed.

4.8.3 Requirements Reasoning

A number of approaches describe reasoning about requirements. Giorgini et al. [93] propose a formal framework for reasoning with goal models. A precise semantics is given for all goal relationships in a qualitative and numerical form. Label propagation algorithms that are shown to be sound and complete with respect to the axiomatization are introduced. Two main limitations are stated. One concerns the definition of contribution links and the labels assignment; the second is that the conflicts relation is not resolved. In general, the idea in [93] is similar to our approach. However, the presented reasoning framework is very specific to goal models. No reasoning facility and tool support is introduced.

Zowghi et al. [268] [267] propose a logical framework for modeling and reasoning about the evolution of requirements. They characterize the properties correctness, completeness, and consistency of requirements in an evolutionary framework. The interaction of consistency and completeness with correctness during requirements evolution is discussed. Duffy et al.

[67] propose a logic-based framework for reasoning about requirements specifications based on goal-tree structures. The framework is based on goal decomposition supported by automated reasoning. Rodrigues et al. [223] propose a framework for the analysis of evolving specifications that enables reasoning in the presence of inconsistency. The work is complementary to our formalization since a tool that translates requirements given in the form of “if then else” rules into the disjunctive normal form for classical logic reasoning and cluster prioritization is provided.

Heitmeyer et al. [113] propose consistency checking in requirements specifications for automatic detection of errors, such as type errors, non-determinism, missing cases, and circular definitions. The technique is based on requirements specifications expressed in the SCR (Software Cost Reduction) tabular notation. A formal requirements model that represents the system to be built as a finite-state automaton is provided. It defines a system state in terms of entities, a condition as a predicate on the system state, and an input event as a change which triggers a new system state. There are some consistency checks derived from the formal requirements model such as type correctness. Contrary to our approach, the formal requirements model requires modeling requirements in a very formal way in order to detect inconsistencies. The main focus is determining inconsistencies among requirements instead of inconsistencies among requirements relations.

Finkelstein et al. [81] [194] describe a technique for inconsistency handling in requirements documents developed using multiple methods and notations for the same system. They combine the ViewPoints framework for perspective development and a logic-based approach to inconsistency handling. Partial specification knowledge in each ViewPoint is translated into first-order logic. Logical inconsistencies are identified. Then, some temporal logic rules are combined with the identified inconsistencies to specify inconsistency handling actions. Hunter et al. [117] present an adaptation of classical logic, which they term quasi-classical (QC) logic that allows reasoning in the presence of inconsistency. This facilitates an analysis of inconsistent information. In our approach, inconsistencies are explained based on the derivation trace of relations.

4.8.4 Tool Support

Some requirements management tools support multiple requirements relation types. The INCOSE management tool survey [124] evaluates these tools according to the criterion traceability analysis, that is, what kinds of trace links the tools provide and what kinds of analyses are performed by the tools. According to the responses of tool vendors in the survey, current industrial tools mostly do not support reasoning about requirements relations.

IBM Rational RequisitePro [119] provides only two relation types between requirements: *traceFrom* and *traceTo*. Since these two relations indicate only the direction, they are very generic relations. In IBM Telelogic Doors [120], there is no predefined requirements relation. The requirements engineer can specify his or her own relation type. However, it is not possible to assign semantics to relation types created by the requirements engineer. The tool provides basic support for change impact analysis. It shows suspected relations when a requirement is updated. Borland Caliber [27] provides only one generic relation type for requirements. This type can be used for different purposes such as part-whole and refinement. The reasoning facilities of the tools IBM Rational RequisitePro, IBM Telelogic Doors, and Borland Caliber are based only on the transitivity property of the relations. These tools do not support consistency checking of the relations.

In TopTeam Analyst [246], there are four relation types. Three of these relations (*traces into*, *impact*, *used in*) are directed and one of the relations (*trace*) is undirected. This undirected relation is considered as a generic relation type for the other relation types. None of these relation types have formal semantics. The tool does not support any reasoning.

We may conclude that some common industrial requirements tools do not support reasoning about relations between requirements or provide formal semantics for relation types.

4.9 Conclusions

There has recently been a growing interest in requirements traceability in the software engineering community and industry. Although considerable research has been devoted to linking requirements in both forward and backward directions, less attention has been paid to linking requirements with other requirements. In this chapter, we focused on requirements and requirements relations from a traceability perspective. A requirements metamodel including relation types with formal semantics was proposed. Existing requirements engineering approaches were surveyed in order to extract the metamodel. We provided semantics of trace relations with formalization in first-order logic. The formalization of relations was used in tool support for inferencing and consistency checking. We illustrated the approach and the tool in the Course Management System requirements document.

The usage of the formal semantics of relation types enables new relations to be inferred and contradicting relations to be determined in requirements documents. There are still open issues. In some cases, relations do not cause any contradiction but violate some of the

constraints in the requirements engineering domain such as “every non-functional requirement should be related with at least one functional requirement”. These constraints may be valid only for a specific requirements engineering approach like goal-oriented requirements engineering. OCL could be used in order to specify these kinds of constraints in the requirements metamodel. However, further research is needed to specify these constraints. Apart from specifying constraints, there might be updates in the requirements metamodel. In the formalization of relations, we stated that the *refines* and *contains* relations imply the *requires* relation. This might be interpreted as a specialization relation between the *requires*, *refines*, and *contains* relations.

Our approach uses the semantic web technologies OWL and Jena instead of MDE technologies such as model transformation languages and engines. OWL and Jena directly support inferencing by using basic properties like symmetry and transitivity. In contrast, in model transformation languages, we have to encode all basic properties and the logic behind them in order to have the same inferencing capability.

Our current support is for textual requirements. We do not have any support for other requirements artifacts like use case or activity diagrams. We improved the usability of the tool for large models with the visual editor which enables selecting requirements to be shown. However, there is still a need to test the usability of the tool for large requirements documents.

In [96], we presented an approach for reusing the formalization of requirements relations for customization of the requirements metamodel. The main focus of the work in [96] is to customize the core requirements metamodel and to apply the inference rules written for the core relations to the customized relations. We showed how we could benefit from this approach by applying it to current requirements modeling approaches like SysML. Our tool needs to be extended to support this customization.

The requirements attributes like priority and status can be included in our reasoning engine. For instance, we may define the constraint that a requirement cannot require another requirement whose priority is lower.

In this chapter we answered *Research Question 4* raised in Chapter 1: *How to model requirements, software architecture and traces with their semantics for change management? What aspects of requirements, software architecture and traces should be modeled and how? How can we use the modeled aspects to reason about requirements, software architecture and traces?* The entities *Requirement* and *Relation* in the requirements metamodel are the aspects of requirements to be modeled. These entities with

their semantics are used in inferencing and consistency checking to reason about requirements.

The results in this chapter like requirements relation types, relation semantics, inferencing and consistency checking is the input for change impact analysis in requirements models. Chapter 5 presents an approach for using requirements relations and their semantics for change impact analysis in requirements models. TRIC is extended with features in order to apply semantics of relations in change impact analysis. For the evolution of requirements, we want to analyze the impact of requirements changes on software architectures. Chapter 6 defines trace relations and their semantics in order to link requirements models to software architecture models with a similar approach presented in this chapter.

Chapter 5

5 Change Impact Analysis in Requirements Models

In this chapter, we provide an approach for change impact analysis in requirements models by using formal semantics of requirements relations and requirements change types. The classification of requirements changes is based on the structure of a textual requirement with formal semantics. The formalization of requirements relations and changes is used for propagating proposed changes and consistency checking of proposed changes in requirements models. Tool support for the approach is an extension of our Tool for Requirements Inferencing and Consistency Checking (TRIC). The main features of the tool are proposing changes, propagating proposed changes, checking consistency of proposed changes, and generating decision trees for reasoning about proposed changes. We illustrate our approach in an example which shows that the formal semantics of requirements relations and change classification provides more precise change impact analysis in requirements models.

5.1 Introduction

In Chapter 3 we analyzed the impacts explosion problem. It is observed that additional semantic information for requirements should be employed to increase the accuracy of impact analysis in requirements models. Chapter 4 focused on relations between requirements in requirements models (see Figure 5.1). Formal definitions of the relation types were provided in order to enable reasoning about requirements relations.

When a change is introduced to a requirement, the requirements engineer determines if there is any impacted requirement. By using only the transitive closure of relations, the requirements engineer may conclude that all requirements in the model are impacted. Without any additional semantic information about the requirements relations and change,

he may have to analyze the whole requirements model for a single change. Furthermore, without considering semantics, change impact analysis may produce high numbers of false positives and false negatives. Consequently, the cost of implementing a change may become several times higher than expected. Assume that a change is introduced to R_n in Figure 5.1. The introduced change in R_n can be traced to other requirements R_1 , R_2 , and R_3 by following the requirements relations. The requirements engineer has to inspect all these requirements to identify what to change in the requirements model.

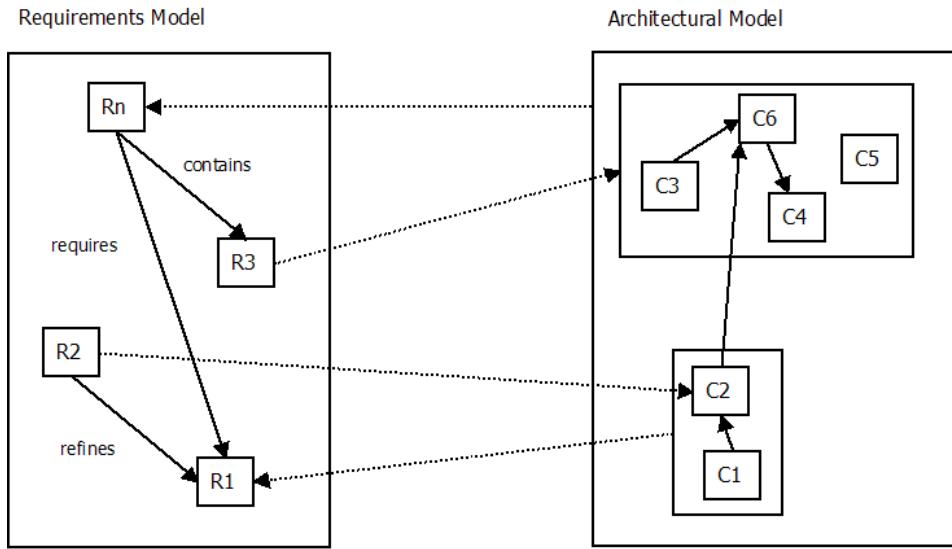


Figure 5.1 Requirements and Architectural Models Showing Within-model and Between-model Trace Relations

In this chapter we provide a change impact analysis approach in requirements models based on formal semantics of requirements relations and requirements change types. Our approach increases the accuracy of impact analysis since it provides change alternatives in change propagation, elimination of false positives and consistency checking of changes.

The classification of requirements changes is based on the structure of a textual requirement. Changes are formalized by giving their effects in terms of formula changes in the requirement. The rationale of changes affects the impact of a change. The rationale of changes is formalized in terms of formula changes in the requirements model. The formalization of requirements relations, changes and change rationale is used for propagating and consistency checking of proposed changes. Here, *propagating proposed changes* is the activity of deducing new proposed changes for requirements related to the requirement having a proposed change. *Consistency checking* is the activity of identifying the proposed changes whose existence may cause a contradiction. Change alternatives in change propagation and

consistency rules for proposed changes are determined based on the semantics of change types, requirements relations and rationale of changes.

TRIC supports change impact analysis in requirements models. The tool supports three activities for impact analysis. First, the requirements engineer proposes changes according to the change classification before implementing the actual changes. Second, the requirements engineer identifies the propagation of the changes to related requirements. Third, possible contradicting changes are identified.

In this chapter we answer *Research Question 5* raised in Chapter 1: *How can a change in a requirement be propagated to other requirements and to software architecture? How can we support the requirements engineer and software architect for performing changes? How can we formally check if the evolved architecture satisfies evolved requirements? How can we become sure that traces are up-to-date?* With the approach for change impact analysis in requirements models we address the issues about propagation of changes from a requirement to other related requirements.

The chapter is structured as follows. Section 5.2 describes the approach. Section 5.3 presents classification of requirements changes with formal semantics. In Section 5.4, we describe the use of the formalization of requirements relations and requirements change types for change propagation and change consistency checking. The approach is discussed for the open issues in Section 5.5. Section 5.6 gives details about the tool support. Section 5.7 illustrates the approach by an example. Section 5.8 evaluates the approach. Section 5.9 summarizes the related work, and Section 5.10 concludes the chapter.

5.2 Approach

We provide an approach for more precise change impact analysis in requirements models by using formal semantics of requirements relations and requirements change types. We rely on the previously defined requirements metamodel with formal semantics. In addition, in this chapter the followings are elaborated:

- **Classification of requirements changes.** To determine the granularity of change that can be applied to requirements, we use the structure of a textual requirement. With this structure, parts of the requirement to which a change is proposed are identified (Section 5.3).
- **Semantics of requirements changes.** Changes are formalized by giving their effects in terms of formula changes in the requirement (Section 5.3).

- **Rationale of requirements changes.** Requirements changes might happen because of different reasons. Rationale of changes affects the change impact. Rationale of changes is formalized in terms of formula changes in the requirements model (Section 5.3).
- **Change propagation and change consistency checking.** The approach identifies change alternatives in the propagation and consistency rules for proposed changes. Change alternatives and consistency rules are determined based on the semantics of change types, requirements relations and rationale of changes (Section 5.4).

We provide tool support and illustrate the feasibility of our approach in an example.

- **Tool support.** We describe the design and implementation of a prototype tool for proposing changes, propagating proposed changes, checking consistency of proposed changes, and implementing proposed changes in the requirements model (Section 5.6).
- **Running example.** The approach is illustrated with an example through the whole chapter (Section 5.7 is a complete example section). The example is about requirements for the Course Management System (CMS) that is also used in Chapter 4. Part of this document is given in Appendix B.

5.3 Classification of Changes in Requirements

In this section, the structure of a textual requirement is mapped to our formalization of a requirement. Requirements changes are classified based on the textual requirement structure. Then, change types are formalized by giving their effects in terms of formula changes in the requirement. We discuss rationale of changes at the end of the section.

5.3.1 Structure of a Textual Requirement

We need to consider the structure of a requirement to determine the granularity of changes that can be applied. Heninger [114] mentions about six criteria which a software requirements document should satisfy:

- It should only specify external system behavior
- It should specify constraints on the implementation
- It should be easy to change

- It should serve as a reference document for system maintainers
- It should record forethought about the life-cycle of the system
- It should characterize acceptable responses to undesired events

The last four criteria can be regarded as quality criteria for the requirements document. The first two criteria explicitly mention external behaviour and constraints on this behaviour respectively.

Wasson [258] further refines external behavior and constraints in order to explain the structure of a textual requirement. He states that a textual requirement should be interpreted by identifying key elements of the requirement, the so-called *requirement primitives*. The requirement primitives in [258] are the following:

- Capability to be provided
- Relational operators
- Thresholds, boundary constraints, tolerances or conditions

Each requirement describes one or more capability that the system should provide. This is the main functionality. This functionality can be further refined by adding additional information which makes the capability more specific. It is done by thresholds, boundary constraints and other limitations such as tolerances. Compared to Heninger, Wasson explains in further detail how a threshold is related to a capability. This is done through the relational operator, which describes how additional information is related to the capability. Using Wasson's primitives, we present the structure of a textual requirement with a UML diagram (see Figure 5.2).

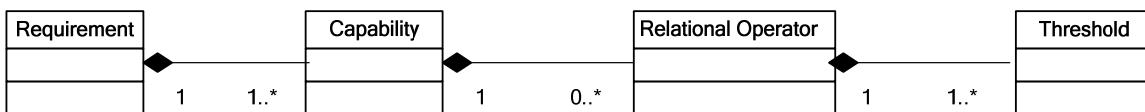


Figure 5.2 Wasson's Primitives for Structure of a Textual Requirement

Example: Structure of Requirement based on Wasson's Primitives

Consider the following requirement.

R98: The system shall allow only the administration to create new courses.

We give the following structure of R98 by using Wasson's primitives in Figure 5.2:

Capability: The system shall provide the functionality of creating new courses

Relational operator: Limited by user type

Threshold: Only by the administration

The definition of a requirement used in Chapter 4 is that “a textual requirement is a description of a property or properties which must be exhibited by the system”. The notion of Property corresponds to the capability where relational operator and threshold in Wasson’s definitions can be classified as constraints over properties (see Figure 5.3).

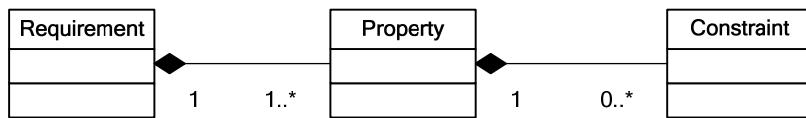


Figure 5.3 Structure of a Textual Requirement based on the Definition of a Requirement in Chapter 4

In Chapter 4, we formalized requirements as formulas in conjunctive normal form (CNF). The properties and constraints in the requirement can be mapped to any conjunct in conjunctive normal form of P. The mapping depends on the interpretation of the requirement as a formula.

Example: Structure of Requirement based on the Definition of a Requirement in Chapter 4

We explain the structure of a textual requirement with the following example.

R98: The system shall allow only the administration to create new courses.

We give the following structure of R98 by using structure of a textual requirement in Figure 5.3:

Property: The system shall provide the functionality of creating courses to only the administration

Constraint: Only the administrator users can create courses

We formalize R98 follows:

$$(43) \quad P_{98} = \forall x \forall y (\neg \text{courses}(x) \vee \neg \text{numbers}(y) \vee \text{create}(x, y, z))$$

where x is a universally quantified variable for the courses, y is a universally quantified variable for the number of students registered to the course, and z is a free variable for the administrators of the system who creates the courses. Let $\mathcal{P} \triangleq \{\text{create}, \text{courses}, \text{numbers}\}$, where *create* is a predicate with three arguments; and where *courses* and *numbers* are predicates with one argument.

Both the *property* and its *constraint* in R98 are mapped to the conjunct $(\neg \text{courses}(x) \vee \neg \text{numbers}(y) \vee \text{create}(x, y, z))$. Different mappings can be defined with different formulas for R98. As a second encoding we formalize R98 as follows:

$$(44) \quad P_{98} = \forall x \forall y ((\neg \text{courses}(x) \vee \neg \text{numbers}(y) \vee \text{create}(x, y)) \wedge \text{administrator}(z))$$

where x is a universally quantified variable for the courses, y is a universally quantified variable for the number of students registered to the course, and z is a free variable for the administrators of the system who creates the courses. Let $\mathcal{P} \triangleq \{\text{create}, \text{courses}, \text{numbers}, \text{administrator}\}$, where *create* is a predicate with three arguments; and where *administrator*, *courses* and *numbers* are predicates with one argument.

The *property* in R98 is mapped to the conjuncts $(\neg \text{courses}(x) \vee \neg \text{numbers}(y) \vee \text{create}(x, y))$ and $(\text{administrator}(z))$ while its constraint is mapped to the conjunct $(\text{administrator}(z))$.

5.3.2 Change Types for Requirements Models

Change types for requirements models are derived from the structure in Figure 5.3 and from the requirements metamodel in Chapter 4. Table 5.1 gives the requirements change types.

Table 5.1 Requirements Change Types

Change Types
• Add a New Requirements Relation
• Delete Requirements Relation
• Update Requirements Relation
• Add a New Requirement
• Delete Requirement
• Update Requirement
○ Add Property to Requirement
○ Add Constraint to Property of Requirement
○ Change Property of Requirement
○ Change Constraint of Property of Requirement
○ Delete Property of Requirement
○ Delete Constraint of Property of Requirement

The first five change types in Table 5.1 are obvious manipulations over the requirements model. The subtypes of ‘Update Requirement’ are obtained from the structure of a textual requirement in Section 5.3.1. We formalize only the subtypes of ‘Update Requirement’ by giving their effects in terms of formula changes in the requirement.

Add a New Requirements Relation

A new requirements relation is added between two requirements R_i and R_k .

Delete Requirements Relation

A requirements relation between two requirements R_i and R_k is removed.

Update Requirements Relation

The type or direction of a requirements relation between two requirements R_i and R_k is changed.

Add a New Requirement

Create a new requirement R to be added to the requirements model.

Delete Requirement

Delete a requirement R from the requirements model.

Update Requirement

We use the symbol \mapsto , to denote updates in requirements in the following way: $R \mapsto R'$ denotes a change where R is the requirement before the change and R' is the requirement after the change. Change types are denoted by using a notation over the symbol \mapsto . Update of a requirement R is done:

- By adding a property pt to the requirement R , denoted as $R \xrightarrow{+pt} R'$.
- By deleting a property pt of the requirement R , denoted as $R \xrightarrow{-pt} R'$.
- By changing a property pt of the requirement R with a property pt' , denoted as $R \xrightarrow{pt \mapsto pt'} R'$.
- By adding a constraint ct to a property pt of the requirement R , denoted as $R \xrightarrow{+ct} R'$.
- By deleting a constraint ct of a property pt of the requirement R , denoted as $R \xrightarrow{-ct} R'$.
- By changing a constraint ct of a property pt of the requirement R with a constraint ct' , denoted as $R \xrightarrow{ct \mapsto ct'} R'$.

We assume that the change ‘Update Requirement’ always changes the set of systems that satisfy the properties in the updated requirement. For instance, for adding a property p_t to the requirement R we assume that the added property p_t is always different than the existing properties in the requirement R . There is always a system s that satisfies p_t and does not satisfy the existing properties in the requirement R . Therefore, the set of systems that satisfy the requirement R is different after adding the property p_t to the requirement R .

In the following we describe the effect of the changes over the formulas.

Add Property to Requirement

Let R be the requirement before adding the property p_t , and R^l be the requirement after adding the property p_t . P and P^l are formulas for R and R^l . P is in conjunctive normal form as follows:

$$(45) \quad P = \forall x (p_1 \dots p_i); i \geq 1$$

$R \xrightarrow{+p_t} R^l$ iff P^l is derived from P such that the following two statements hold:

$$(46) \quad P^l = P \wedge P_{p_t}$$

$$(47) \quad (\neg(P \rightarrow P^l)) \text{ is satisfiable}$$

where P_{p_t} denotes the property that is captured in p_t

For the formulas P and P_{p_t} , if any variable universally quantified in one of the formulas appears free in the second formula, the free variable is renamed. If any variable in P appears in P_{p_t} with a different valuation, the variable in P_{p_t} is renamed. Please note that if the requirements R^l and R are written as formulas $\forall x \varphi_1$ and $\forall x \varphi_2$ with φ_1 and φ_2 in CNF and

P_{p_t} is expressed as $\forall x \psi$ with ψ in CNF, we understand the following: $R \xrightarrow{+p_t} R^l$ iff $(P^l = \forall x (\varphi_2 \wedge \psi))$, and $(\neg(\forall x (\varphi_2 \rightarrow \varphi_1)))$ is satisfiable.

From the definition we conclude that $(P^l \rightarrow P)$ and $(P^l \rightarrow P_{p_t})$ hold for every model where $R \xrightarrow{+p_t} R^l$. We assume that the change ‘Update Requirement’ always changes the set of systems that satisfy the properties in the updated requirement. P and P_{p_t} always describe different system properties. S^l is a proper subset of S ($S^l \subset S$) where S^l is the set of systems that satisfy R^l and S is the set of systems that satisfy R .

Example: Add Property to Requirement

Consider the following requirement.

R62: The system shall allow lecturers to specify enrolment policies based on grade.

We formalize R62 as follows

$$(48) \quad P_{62} = \text{allow}(\text{grade_enrl_policy}).$$

where *grade_enrl_policy* is a constant. We add a property *pt* to the requirement R62 ($R62 \xrightarrow{+pt} R62'$) where we have a new requirement as follows.

R62^l: The system shall allow lecturers to specify enrolment policies based on grade and first come-first serve.

We formalize R62^l as follows

$$(49) \quad P_{62^l} = \text{allow}(\text{grade_enrl_policy}) \wedge \text{allow}(\text{fcfs_enrl_policy})$$

where *grade_enrl_policy*, and *fcfs_enrl_policy* are constants. We have the following:

$$(50) \quad P_{62^l} = P_{62} \wedge \text{allow}(\text{fcfs_enrl_policy})$$

Let $\mathcal{F} \triangleq \{\text{fcfs_enrl_policy}, \text{department_enrl_policy}, \text{grade_enrl_policy}\}$ and $\mathcal{P} \triangleq \{\text{allow_policy}\}$, where *fcfs_enrl_policy*, *department_enrl_policy* and *grade_enrl_policy* are constant symbols; and where *allow_policy* is a predicate with one argument. We choose as a model \mathcal{M} the following:

- $A \triangleq \{\text{fcfs_enrolment_policy}, \text{department_enrolment_policy}, \text{grade_enrolment_policy}\}$
- $\text{fcfs_enrl_policy}^{\mathcal{M}} \triangleq \text{fcfs_enrolment_policy}$
- $\text{department_enrl_policy}^{\mathcal{M}} \triangleq \text{department_enrolment_policy}$
- $\text{grade_enrl_policy}^{\mathcal{M}} \triangleq \text{grade_enrolment_policy}$
- $\text{allow_policy}^{\mathcal{M}} \triangleq \{\text{department_enrolment_policy}, \text{grade_enrolment_policy}\}$

Then we have the following:

$$(51) \quad \mathcal{M} \models \neg(\text{allow_policy}(\text{grade_enrl_policy}) \rightarrow (\text{allow_policy}(\text{grade_enrl_policy}) \wedge \text{allow_policy}(\text{fcfs_enrl_policy})))$$

R62¹ states that the system shall allow lecturers to specify two different enrollment policies. The requirement can be interpreted as two different properties for the system, like *specifying enrolment policies based on grade*, and *specifying enrolment policies based on first come first serve*. R62 states only one of these properties, which is *specifying enrolment policies based on grade*. Therefore, the property *specifying enrolment policies based on first come first serve* is added to the requirement R62.

Delete Property of Requirement

Let R be the requirement before deleting the property p_t , and R^l be the requirement after deleting the property p_t . P and P^l are formulas for R and R^l . P is in conjunctive normal form as follows:

$$(52) \quad P = \forall x ((p_1 \dots p_n) \wedge (q_1 \dots q_m)); \quad m, n \geq 1$$

$R \xrightarrow{-p_t} R^l$ iff P^l is derived from P such that the following two statements hold:

$$(53) \quad P^l = \forall x (p_1 \dots p_n); \quad n \geq 1$$

$$(54) \quad (\neg(P^l \rightarrow P)) \text{ is satisfiable}$$

where $\forall x (q_1 \dots q_m)$ denotes the property that is captured in p_t .

If every bounded occurrence of a variable is removed by deleting the property, then the quantifier for the variable is removed as well. Please note that if the requirement R is written as a formula $\forall x(\varphi \wedge \psi)$ with $(\varphi \wedge \psi)$ in CNF and P_{pt} (for the property captured in p_t) is expressed as $\forall x \psi$ with ψ in CNF, we understand the following: $R \xrightarrow{-p_t} R^l$ iff $(P^l = \forall x \varphi)$, and $(\neg(\forall x(\varphi \rightarrow (\varphi \wedge \psi))))$ is satisfiable.

From the definition we conclude that $(P \rightarrow P^l)$ and $(P \rightarrow P_{pt})$ hold for every model where $R \xrightarrow{-p_t} R^l$. We assumed that the change 'Update Requirement' always changes the set of systems that satisfy the properties in the updated requirement. $\forall x(p_1 \dots p_n)$ and $\forall x(q_1 \dots q_m)$ always describe different system properties. S is a proper subset of S^l ($S \subset S^l$) where S is the set of systems that satisfy R and S^l is the set of systems that satisfy R^l .

Change Property of Requirement

Let R be the requirement before changing the property p_t with the property p_t' , and R^l be the requirement after changing the property p_t with the property p_t' . P and P^l are formulas for R and R^l . P is in conjunctive normal form as follows:

$$(55) \quad P = \forall x ((p_1 \dots p_n) \wedge (q_1 \dots q_m)); \quad n \geq 1, m \geq 0$$

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

$$(56) \quad P^l = \forall x ((t_1 \dots t_z) \wedge (q_1 \dots q_m)); \quad z \geq 1, m \geq 0$$

where $\forall x(p_1 \dots p_n)$ denotes the property captured in pt and $\forall x(t_1 \dots t_z)$ denotes the property captured in pt^l .

If every bounded occurrence of a variable is removed by changing the property, then the quantifier for the variable is removed as well. For the formulas $\forall x(t_1 \dots t_z)$ and $\forall x(q_1 \dots q_m)$, if any variable universally quantified in one of the formulas appears free in the second formula, the free variable is renamed. If any variable in $\forall x(q_1 \dots q_m)$ appears in $\forall x(t_1 \dots t_z)$ with a different valuation, the variable in $\forall x(t_1 \dots t_z)$ is renamed.

Add Constraint to Property of Requirement

Let R be the requirement before adding the constraint $c\ell$ to the property pt , and R^l be the requirement after adding the constraint $c\ell$ to the property pt . P and P^l are formulas for R and R^l . P is in conjunctive normal form as follows:

$$(57) \quad P = \forall x ((p_1 \dots p_n) \wedge (q_1 \dots q_m)); \quad n \geq 1, m \geq 0$$

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

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

$$(58) \quad P^l = \forall x ((p_1^l \dots p_n^l) \wedge (q_1 \dots q_m)); \quad n \geq 1, m \geq 0$$

$$(59) \quad (\neg(\forall x(p_j \rightarrow p_j^l))) \text{ is satisfiable for all } j \in 1..n$$

For the formulas $\forall x(p_1^l \dots p_n^l)$ and $\forall x(q_1 \dots q_m)$, if any variable universally quantified in one of the formulas appears free in the second formula, the free variable is renamed. If any variable in $\forall x(q_1 \dots q_m)$ appears in $\forall x(p_1^l \dots p_n^l)$ with a different valuation, the variable in $\forall x(p_1^l \dots p_n^l)$ is renamed.

This change is similar to refining a requirement (see the *refines* relation in Chapter 4). The idea behind the change is to make the requirement more restrictive by adding constraint. Therefore, the requirement after the change is a refinement of the requirement before the change. From the definition of the *refines* relation we conclude that $(P^l \rightarrow P)$ holds for every

model where $R \xrightarrow{+ct} R^l$ and $(\neg(P \rightarrow P^l))$ is satisfiable. ($S^l \subset S$) where S^l is the set of systems that satisfy R^l and S is the set of systems that satisfy R . Extensionally, the changes ‘Add Property to Requirement’ and ‘Add Constraint to Property of Requirement’ are the same since these two changes cause a proper subsetting between the sets of systems ($S^l \subset S$). Intensionally, they are different, i.e. they have different effects on formulas. The different effects on formulas are the reason of having different propagations for these changes (see Section 5.4 for change propagation).

Example: Add Constraint to Property of Requirement

Consider the following requirement.

R7: The system shall provide a messaging facility.

We add a constraint ct to the property pt of the requirement R7 ($R7 \xrightarrow{+ct} R7^l$) where we have a new requirement as follows.

R7^l: The system shall allow messages to be sent to individuals, teams, or all course participants at once.

We formalize the requirements R7 and R7^l as follows:

$$(60) \quad P_7 = \text{provide_msg}(x)$$

$$(61) \quad P_7^l = \text{course_msg}(x)$$

where x is a free variable over the values in A . Let $\mathcal{P} \stackrel{\text{def}}{=} \{\text{provide_msg}, \text{course_msg}\}$ where *provide_msg* and *course_msg* are predicates with one argument. From the domain knowledge we know that the following statement is valid for all models:

$$(62) \quad \text{course_msg}^M \subseteq \text{provide_msg}^M$$

We choose as a model M the following:

- $A \stackrel{\text{def}}{=} \{\text{individual_msg}, \text{team_msg}, \text{participant_msg}, \text{lecturer_msg}\}$
- $\text{provide_msg}^M \stackrel{\text{def}}{=} \{\text{individual_msg}, \text{team_msg}, \text{participant_msg}, \text{lecturer_msg}\}$
- $\text{course_msg}^M \stackrel{\text{def}}{=} \{\text{individual_msg}, \text{team_msg}, \text{participant_msg}\}$

Then we have the following:

$$(63) \quad \mathcal{M} \models \ell \text{ course_msg}(x) \rightarrow \text{provide_msg}(x)$$

The relation $\text{course_msg}^{\mathcal{M}}$ is a subset of the relation $\text{provide_msg}^{\mathcal{M}}$. Therefore, $(\text{course_msg}(x) \rightarrow \text{provide_msg}(x))$ holds for each ℓ with the model \mathcal{M} . $(\neg(\text{provide_msg}(x) \rightarrow \text{course_msg}(x)))$ is satisfiable like in the following:

$$(64) \quad \mathcal{M} \models \ell_{[x \mapsto \text{lecturer_msg}]} (\neg(\text{provide_msg}(x) \rightarrow \text{course_msg}(x)))$$

R7 states only the need for a messaging property in the system. However, R^{7l} explains the details of the messaging property: the messaging shall allow messages to be sent to individuals, teams, or all course participants at once, excluding lecturers.

Please note that as we saw in the example in Section 5.3.1 there might be other encodings of R7.

Delete Constraint of Property of Requirement

Let R be the requirement before deleting the constraint ℓ from the property pt , and R^l be the requirement after deleting the constraint ℓ from the property pt . P and P^l are formulas for R and R^l . P is in conjunctive normal form as follows:

$$(65) \quad P = \forall x ((p_1^l \dots p_n^l) \wedge (q_1 \dots q_m)); \quad n \geq 1, m \geq 0$$

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

$R \stackrel{-\ell}{\mapsto} R^l$ iff P^l is derived from P by replacing every p_j^l in P with p_j for $j \in 1..n$ such that the following two statements hold:

$$(66) \quad P^l = \forall x ((p_1 \dots p_n) \wedge (q_1 \dots q_m)); \quad n \geq 1, m \geq 0$$

$$(67) \quad (\neg(\forall x (p_j \rightarrow p_j^l))) \text{ is satisfiable for all } j \in 1..n$$

If every bounded occurrence of a variable is removed by deleting the constraint of the property, then the quantifier for the variable is removed as well. For the formulas $\forall x (p_1 \dots p_n)$ and $\forall x (q_1 \dots q_m)$, if any variable universally quantified in one of the formulas appears free in the second formula, the free variable is renamed. If any variable in $\forall x (q_1 \dots q_m)$ appears in $\forall x (p_1 \dots p_n)$ with a different valuation, the variable in $\forall x (p_1 \dots p_n)$ is renamed.

This change is similar to refining a requirement (see the *refines* relation in Chapter 4). The idea behind the change is to make the requirement less restrictive by removing constraint. From the definition of the *refines* relation we conclude that $(P \rightarrow P^l)$ holds for every model

where $R \xrightarrow{-ct} R^l$ and $(\neg(P^l \rightarrow P))$ is satisfiable. ($S \subset S^l$) where S is the set of systems that satisfy R and S^l is the set of systems that satisfy R^l . Extensionally, the changes ‘Delete Property of Requirement’ and ‘Delete Constraint of Property of Requirement’ are the same since for these two changes there is a proper subsetting between the sets of systems ($S \subset S^l$). Intensionally, they are different, i.e. they have different effects on formulas. The different effects on formulas are the reason of having different propagations for these two changes (see Section 5.4 for change propagation).

Change Constraint of Property of Requirement

Let R be the requirement before changing the constraint ct with the constraint ct^l , and R^l be the requirement after changing the constraint ct with the constraint ct^l . P and P^l are formulas for R and R^l . P is a formula in conjunctive normal form as follows:

$$(68) \quad P = \forall x ((p_1 \dots p_n) \wedge (q_1 \dots q_m)); \quad m, n \geq 1$$

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

$$(69) \quad P^l = \forall x ((t_1 \dots t_z) \wedge (q_1 \dots q_m)); \quad m, z \geq 1$$

where $\forall x(p_1 \dots p_n)$ denotes the constraint captured in ct and $\forall x(t_1 \dots t_z)$ denotes the constraint captured in ct^l .

If every bounded occurrence of a variable is removed by changing the constraint of the property, then the quantifier for the variable is removed as well. For the formulas $\forall x(t_1 \dots t_z)$ and $\forall x(q_1 \dots q_m)$, if any variable universally quantified in one of the formulas appears free in the second formula, the free variable is renamed. If any variable in $\forall x(q_1 \dots q_m)$ appears in $\forall x(t_1 \dots t_z)$ with a different valuation, the variable in $\forall x(t_1 \dots t_z)$ is renamed.

The effects of the changes “Change constraint” and “Change Property” in terms of formula changes in the formalization are the same. In these two changes some of the conjuncts in the formula are replaced by new conjuncts.

5.3.3 Rationale of Changes

The semantics of requirements changes in Section 5.3.2 does not explain why a change needs to be performed in the requirements model, that is, what is the rationale of changes. The impact of changes depends on their rationale. For instance, the requirements engineer may delete a property of a requirement because this property is not required any more from business/stakeholder point of view. The property may be in other requirements in the model and it also has to be deleted from them. The requirements engineer may delete a property of

a requirement in the requirements model to improve the structure of the model without modifying overall system properties. This property still must hold in the requirements model after the change. The property has to be kept at least in one of the requirements in the model. Therefore, we need to know rationale of requirements changes in order to determine the impact of changes in the whole requirements model. We classify rationale of requirements changes as *refactoring* and *domain changes*.

Buckley et al. [40] classifies changes in general as *semantics-preserving* and *semantics-modifying*. However, they focus more on semantics of software components, such as type hierarchy, scoping, visibility, accessibility, and overriding relationships, rather than changes in requirements. We adapt the classification proposed by Buckley for requirements changes. Van Lamsweerde [151] introduces requirements description qualities such as good structuring and modifiability. The requirements engineer may change the requirements model to improve the quality of requirements description. For instance, a requirement may be decomposed to multiple requirements. These changes are *semantics-preserving* according to [40] and we consider their rationale as *refactoring* (see [82] for refactoring). Evolution of requirements also fosters changes to the requirements model. We name these changes and their rationale *domain changes*. With the term ‘domain’ we mean problem/business domain. Consider a requirements model that contains a set of requirements for online banking in Europe. Here, the domain is banking and a change request for adapting the system to the USA is received. Then, all currency requirements in the domain of banking are changed and these changes should be reflected in the requirements model.

In order to formalize domain changes and refactoring, we first formalize the requirements model in the following.

We define a requirements model RM as a property (or properties). We express the property (or properties) as a formula P_{RM} in CNF. P_{RM} can be represented in a conjunctive normal form (CNF) in the following way:

$$(70) \quad P_{RM} = \forall x (p_1 \wedge \dots \wedge p_n), \text{ where } n \geq 1 \text{ and } p_n \text{ is disjunction of literals}$$

The requirements model RM from Chapter 4 contains a set of requirements formalized as R_1, R_2, \dots, R_k where $k \geq 1$. $P_1, P_2, P_3, \dots, P_k$ are formulas for R_1, R_2, \dots, R_k in conjunctive normal form. Therefore, P_{RM} can also be represented in the following way.

$$(71) \quad P_{RM} = P_1 \wedge P_2 \wedge \dots \wedge P_k$$

Please note that if the requirements R_1, R_2, \dots, R_k are written as formulas $\forall x\varphi_1, \forall x\varphi_2, \dots, \forall x\varphi_k$ with $\varphi_1, \varphi_2, \dots, \varphi_k$ in CNF, we have the following: ($P_{RM} = \forall x(\varphi_1 \wedge \varphi_2 \wedge \varphi_3 \wedge \dots \wedge \varphi_k)$).

Refactoring

Refactoring is a change (changes) in the requirements model in order to improve the structure of the model without modifying overall system properties [82]. Changes to the model caused by refactoring do not affect the properties in the whole requirements model. We formalize the refactoring in the following.

$RM \mapsto RM'$ denotes a series of changes for model refactoring where RM is the requirement model before the changes and RM' is the requirement model after the changes. P_{RM} and P_{RM}' are formulas for RM and RM' . P_{RM} and P_{RM}' are described in the conjunctive normal form in the following.

$$(72) \quad P_{RM} = P_{RM}' = \forall x (p_1 \wedge \dots \wedge p_n), \text{ where } n \geq 1 \text{ and } p_n \text{ is disjunction of literals}$$

In refactoring, changes to the model do not affect the conjunctive normal form (CNF) of the formula of the requirements model although the CNFs of formulas of some requirements in the model are changed. ($S_{RM} = S_{RM}'$) where S_{RM} is the set of systems that satisfy P_{RM} and S_{RM}' is the set of systems that satisfy P_{RM}' .

Domain Changes

Domain changes are the changes in the requirements model in order to modify overall system properties. Changes to the model caused by domain changes do affect the properties in the whole requirements model and usually alter the set of systems that satisfy the requirements. We formalize the domain changes in the following.

$RM \mapsto RM'$ denotes a series of changes caused by domain change where RM is the requirement model before the changes and RM' is the requirement model after the changes. P_{RM} and P_{RM}' are formulas for RM and RM' . We have the following.

$$(73) \quad \neg \text{equals}(P_{RM}, P_{RM}')$$

If two formulas have the same predicate symbols and arguments, and they both have either negation or not, these two formulas are equal. If there are contradicting requirements in the model, $S_{RM} = 0$ where S_{RM} is the set of systems that satisfy P_{RM} . Domain changes will not

change the set of systems that satisfy the requirements in the model unless the conflicts in the model are resolved. For domain changes ($S_{RM} \neq S_{RM}'$) where $S_{RM} \neq 0$.

Rationale of changes is important since it is a factor in order to determine the change alternatives for change propagation (see the example derivation of change alternatives for change propagation in Section 5.4).

5.4 Change Propagation and Change Consistency Checking

Change propagation aims at deducing new proposed changes based on an initial set. Change consistency checking identifies contradicting proposed changes. We provide change propagation defined as a change impact function. Only domain changes are considered in the approach. Given the type of changes, we individually describe rules to determine the impact of each change type. By using the formal semantics of requirements, relations and changes, it is possible to derive whether or not (possible) impacts are caused by a change. The change impact function takes a change type and a requirement to which the change is introduced as input, and produces a set of decision trees as output. A decision tree contains decisions taken for propagating changes by traversing the requirements model. The following is the definition of the change impact function:

$$\text{impact} : SCT \times SR \times SSRR \rightarrow SSDT$$

where SCT is the set of change types, SR is the set of requirements, $SSRR$ is the set of sets of requirements relations, and $SSDT$ is the set of sets of decision trees for changes.

A decision tree is expressed as a sentence in a language with the following grammar.

```

<DT-C> ::= <Change> | <Change> <And> "(" <DT-C> ")" | 
           <DT-C> <Boolean-Operator> <DT-C> | "(" <DT-C> ")"
<Change> ::= <Change-Type> ID
<Change-Type> ::= "No Impact in" | "Delete Requirement" |
                  "Delete Property of Requirement" |
                  "Delete Constraint of Property of Requirement" |
                  "Add Requirement" | "Add Property to Requirement" |
                  "Add Constraint to Property of Requirement" |
  
```

“Change Property of Requirement” |
 “Change Constraint of Property of Requirement” |
 “Add Relation” | “Delete Relation”
 <Boolean-Operator> ::= <Exclusive-or> | <And>
 <Exclusive-or> ::= “|”
 <And> ::= “&”
 ID denotes identifiers.

The algorithm for the change impact function is based on traversing the requirements model and propagating change from one requirement to another related requirement. The *impact* function propagates change from one requirement to other related requirements and returns the set of decision trees. We give the overview of the algorithm for the change impact function in the following:

```

1 Set sdt = empty-set
2
3 impact(ChangeType c, Requirement r, Set srl): Set {
4
5   Set visited = empty-set           // set of visited requirements
6   DecisionTree dt = empty
7   Requirement rq = empty
8
9   dt = createDT(r, c)
10
11  visited = addVisited(r, visited)
12
13  If (srl is an empty-set) { Return empty-set }
14
15  ForEach relation rl ∈ srl {
16    rq = getRequirement(r, rl)
17
18    If (Not rq ∈ visited) { propagateChange(rq, rl, dt, visited) }
19  }
20
21  Return sdt
22 } //End of the impact function
  
```

The variable declarations are done in line 1 and lines 5-7. *sdt* is a variable for the set of decision trees and it is global for the functions *impact*, *propagateChange* and *expandDecisionTrees*

(see line 1). The requirement to which the change is proposed is the starting requirement (Requirement r in line 3). The algorithm creates a decision tree for each unvisited requirement directly related to the starting requirement (see line 9). Once the algorithm is initiated, only the starting requirement is visited (see line 11). If there is no requirement related to the starting requirement, there is no impacted requirement and the function returns an empty set (see line 13). Each decision tree has a root node including the proposed change and the starting requirement. For each unvisited related requirement, the change is propagated (see lines 15-19). If the related requirement is not visited before, then the $propagateChange$ function is called (see line 18). The $impact$ function returns the set of decision trees (see line 21). The $propagateChange$ function propagates the change from the starting requirement to the unvisited related requirement by expanding the decision tree. The overview of the algorithm for the $propagateChange$ function is the following:

```

1 propagateChange(Requirement rq, Relation rl, DecisionTree dt, Set visited) {
2
3     Set cvisited = empty-set           // copy of the set of visited requirements
4     DecisionTree cdt = empty          // copy of the decision tree
5     Set srl = empty-set               // set of relations
6     Requirement req = empty
7     Relation rlt = empty
8
9     cdt = copyDT(dt)
10
11    expandDecisionTree(cdt, rq, rl)
12
13    cvisited = addVisited(rq, visited)
14
15    srl = getRelations(rq)
16
17    ForEach relation rlt ∈ srl {
18        req = getRequirement(rq, rlt)
19
20        If (Not req ∈ cvisited) { propagateChange(req, rlt, cdt, cvisited) }
21    }
22
23    If No requirement rqt such that
24        (getRelation(rqt, rq) ∈ srl) ∧ (rqt ∉ cvisited) {
25        addDT(cdt, sdt)
26    }
27 } //End of the propagateChange function

```

First the decision tree is copied (see line 9). Alternative proposed changes are identified for the unvisited related requirement. The change alternatives in the propagation are determined

based on the semantics of change type and the requirements relation (see Table 5.2 which is explained later). The copied decision tree is expanded with the change alternatives (see line 11). The requirement to which the change is propagated is marked as visited (see line 13). The algorithm is iterative (see lines 15-21). For each decision tree, the set of visited requirements is copied and the directly related requirement to which the change alternatives are introduced becomes the starting requirement. Changes are propagated for each unvisited requirement directly related to the new starting requirement. If there is more than one unvisited related requirement, the decision tree is copied. If there is no unvisited requirement directly related to the starting requirement, the decision tree is added to the set of decision trees (see lines 23-26). Please note that the *impact* function returns the set of decision trees. The algorithm for the *expandDecisionTree* function is the following:

```

1 expandDecisionTree(DecisionTree dt, Requirement rq, Relation rl) {
2
3     Set sct = empty-set      // set of change types
4
5     ForEach leaf node ln in dt {
6         sct = getChangeTypes(ln, rq, rl)
7
8         ForEach change type ct ∈ sct {
9             addChild(ln, rq, ct)
10        }
11    }
12 } //End of the expandDecisionTree function

```

Decision trees are expanded with a set of alternative proposed changes based on BNF given before. For each leaf node of the decision tree, a set of alternative proposed changes is identified for the unvisited related requirement (see line 6). Each alternative becomes a node in the decision tree (see lines 8-10).

Figure 5.4(a) gives an example requirements model where the change ‘Delete Property of Requirement’ is proposed for requirement R2. Figure 5.4(b) shows the four paths created while the change impact algorithm traverses the requirements model for the proposed change in requirement R2.

Figure 5.5 illustrates the decision trees created for the example model in Figure 5.4(a). The operator *Exclusive-or* in the grammar is represented as branches of the decision trees in Figure 5.5 while the operator *And* in the grammar is the “&” in the nodes of the decision trees.

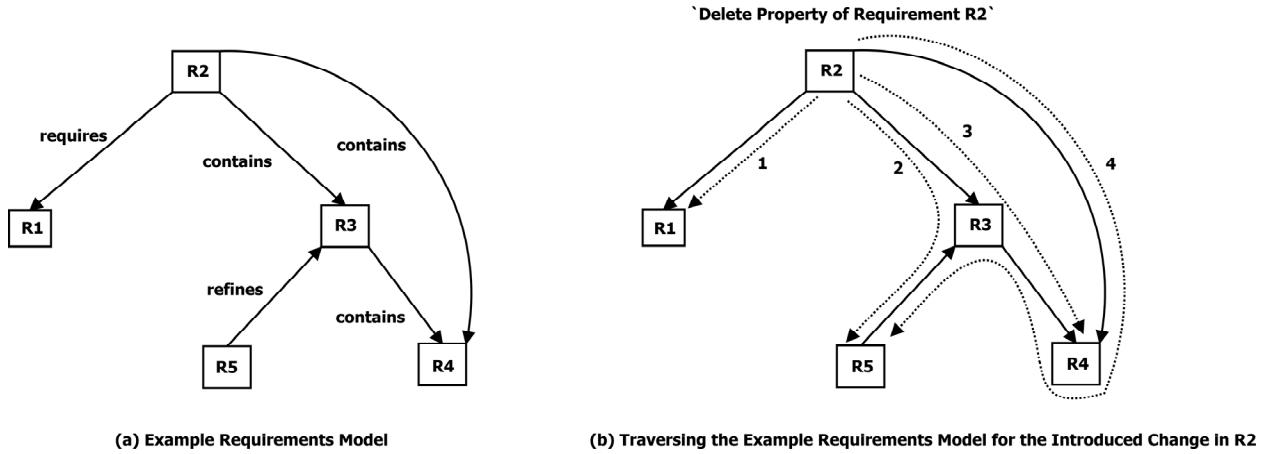


Figure 5.4 Example Requirements Model and Traversing the Model for the Proposed Change

The main steps in the change impact function algorithm are the following:

- *Creating a Decision Tree for Each Unvisited Requirement Related to Starting Requirement* (see the *impact* function). In Figure 5.4(b), the change ‘Delete Property of Requirement’ is introduced to the requirement R2. The algorithm creates a decision tree (*Decision Tree for Path 1*, *Decision Tree for Path 2* and *Decision Tree for Path 4* in Figure 5.5) for each unvisited directly related requirement (R1, R3 and R4 in Figure 5.4(b)). Decision trees have a starting node ‘Delete Property of Requirement R2’.
- *Propagating Change for Each Unvisited Related Requirement* (see the *propagateChange* function). Change alternatives are identified for unvisited requirements (R1, R3 and R4) directly related to R2 in Figure 5.4(b). For instance, R1 is related to R2 through the *requires* relation. The alternatives for propagating the change ‘Delete Property of Requirement R2’ from R2 to R1 are ‘No impact in R1’, ‘Delete Relation’ and ‘Delete R1 & Delete Relation’ (the *Decision Tree for Path 1* in Figure 5.5). These alternatives are given in Table 5.2 where $(R_i \xrightarrow{-pt} R_i')$ and $(R_i \text{ requires } R_k)$.
- *Expanding Decision Tree for Each Unvisited Related Requirement* (see the *expandDecisionTree* function). Each decision tree created for directly related requirements (the *Decision Tree for Path 1* for requirement R2, the *Decision Tree for Path 2* for requirement R3 and the *Decision Tree for Path 4* for requirement R4 in Figure 5.5) is expanded with alternative proposed changes. For instance, the change alternatives ‘No impact in R1’, ‘Delete Relation’ and ‘Delete R1 & Delete Relation’ for R1 become the nodes of the *Decision Tree for Path 1* in Figure 5.5.

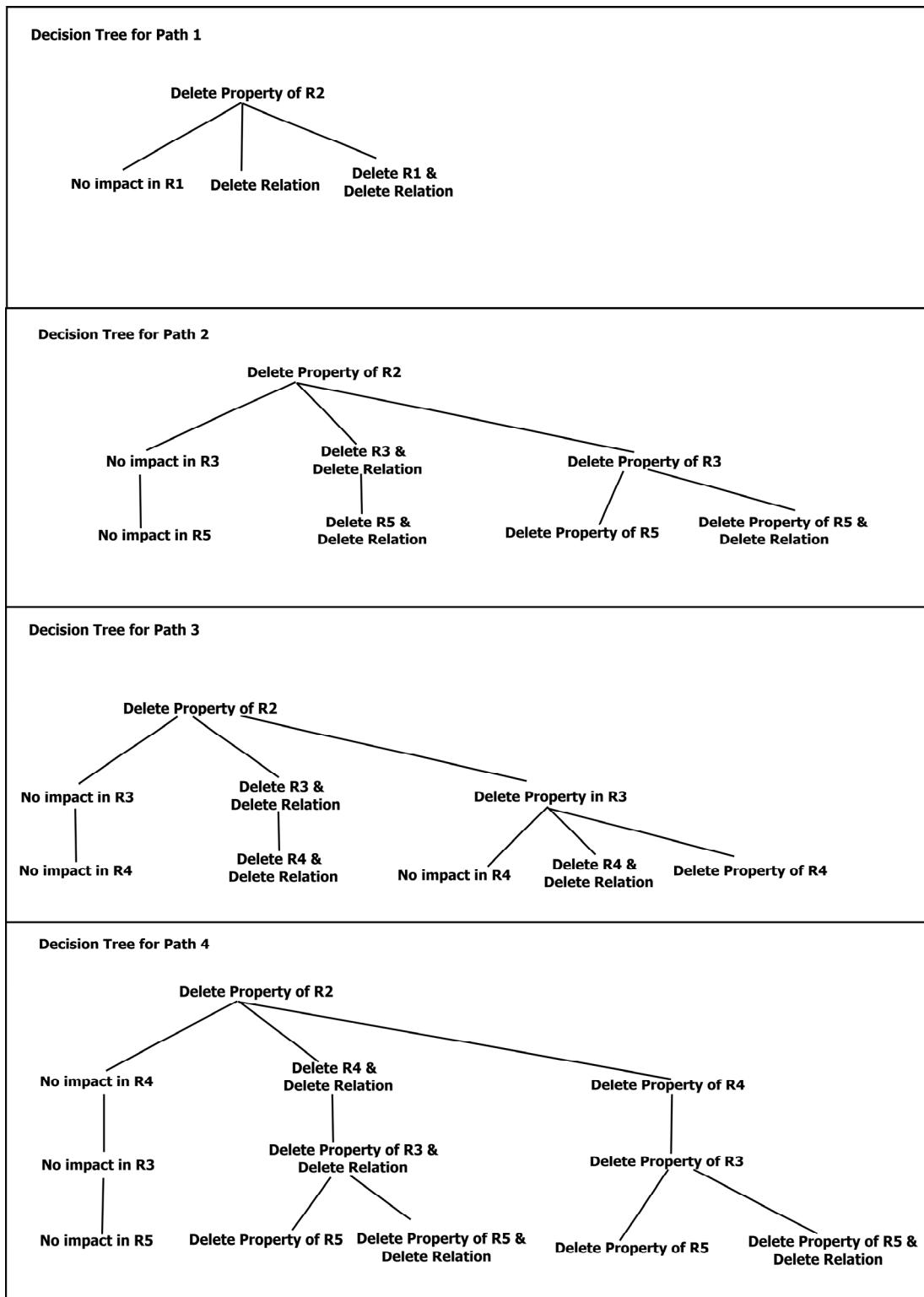


Figure 5.5 Decision Trees for the Example Requirements Model

- *Iterating* (see the *propagateChange* function). Directly related requirements (R_1 , R_3 and R_4 in Figure 5.4(b)) become the starting requirement and the algorithm is reinitiated for each of them. For R_1 , there is no unvisited directly related requirement and the *Decision Tree for Path 1* in Figure 5.5 is not expanded further. For R_3 , there are two unvisited directly related requirements (R_4 and R_5) and the *Decision Tree for Path 2* is copied (see the *Decision Tree for Path 3*). The *Decision Tree for Path 2* is expanded with change alternatives for R_5 and the *Decision Tree for Path 3* is expanded with change alternatives for R_4 .

The output of the change impact function is a set of decision trees that contains all alternatives for a change to be propagated in the whole model. For instance, the output of the change impact function for the proposed change in the example requirements model in Figure 5.4(a) is the set of decision trees in Figure 5.5. The requirements engineer can also select among the change alternatives to propagate the change from one requirement to another step by step. Our tool (see Section 5.6) supports both the decision tree generation and step by step propagation. In the following we explain how change alternatives are derived for change propagation based on the semantics of change types, rationale of changes and requirements relations.

Change Propagation. This is the activity of deducing new proposed changes for requirements related to the requirement which already has a proposed change. The change alternatives are determined based on the semantics of change types, rationale of changes and requirements relations. Table 5.2 gives the change impact alternatives for domain changes. Each cell in the table gives change alternatives in order to propagate the changes in the rows by using the relations in the columns. Please note that both directions of the relations are explored in Table 5.2.

Table 5.2 Change Impact Alternatives for Domain Changes

Changes	Requirements Relation Types				
	R_i contains R_k	R_i refines R_k	R_i partially refines R_k	R_i requires R_k	R_i conflicts R_k
Add R_x	No impact	No impact	No impact	No impact	No impact
Delete Relation	No impact	No impact	No impact	No impact	No impact
Delete R_i	Delete R_k & Delete relation	Delete R_k & Delete relation	Delete property of R_k	Delete relation (Delete R_k & Delete relation)	Delete relation
$R_i \xrightarrow{+pt} R'_j$	No impact	Add property to R_k Delete	Delete relation	No impact	No impact

		relation			
$R_i \xrightarrow{-pt} R'_i$	No impact Delete relation (Delete R_k & Delete relation) Delete property of R_k	Delete property of R_k (Delete property of R_k & Delete relation)	Delete property of R_k	No impact Delete relation (Delete R_k & Delete relation)	No impact Delete relation
$R_i \xrightarrow{pt \rightarrow pt'} R'_i$	No impact Change property of R_k	Change property of R_k (Change property of R_k & Delete relation)	Change property of R_k (Change property of R_k & Delete relation)	No impact Delete relation (Delete R_k & Delete relation)	No impact Delete relation
$R_i \xrightarrow{+ct} R'_i$	No impact Add constraint to property of R_k Delete relation	No impact	No impact	No impact	No impact
$R_i \xrightarrow{-ct} R'_i$	No impact Delete relation Delete constraint of property of R_k (Delete constraint of property of R_k & Delete relation)	No impact Delete relation Delete constraint of property of R_k (Delete constraint of property of R_k & Delete relation)	No impact Delete relation (Delete R_k & Delete relation)	No impact Delete relation	No impact Delete relation
$R_i \xrightarrow{ct \rightarrow ct'} R'_i$	No impact Change constraint of property of R_k	No impact Change constraint of property of R_k	No impact Change constraint of property of R_k	No impact Delete relation (Delete R_k & Delete relation)	No impact Delete relation
Delete R_k	Delete property of R_i	Delete R_i & Delete relation	Delete R_i & Delete relation	Delete relation (Delete R_i & Delete relation)	Delete relation
$R_k \xrightarrow{+pt} R'_k$	Add property to R_i Delete relation	Add property to R_i Delete relation	No impact	No impact	No impact
$R_k \xrightarrow{-pt} R'_k$	Delete property of R_i	Delete property of R_i (Delete property of R_i & Delete relation)	No impact Delete relation Delete property of R_i (Delete R_i & Delete relation)	No impact Delete relation (Delete R_i & Delete relation)	No impact Delete relation
$R_k \xrightarrow{pt \rightarrow pt'} R'_k$	Change property of R_i	Change property of R_i (Change property of R_i & Delete relation)	No impact Change property of R_i (Change property of R_i & Delete relation)	No impact Delete relation (Delete R_i & Delete relation)	No impact Delete relation
$R_k \xrightarrow{+ct} R'_k$	Add constraint to property of R_i Delete relation	Add constraint to property of R_i Delete relation	No impact Add constraint to property of R_i	No impact	No impact
$R_k \xrightarrow{-ct} R'_k$	Delete constraint of property of R_i	Delete constraint of property of R_i (Delete constraint of property of R_i & Delete relation)	No impact Delete constraint of property of R_i (Delete constraint of property of R_i & Delete relation)	No impact Delete relation (Delete R_i & Delete relation)	No impact Delete relation
$R_k \xrightarrow{ct \rightarrow ct'} R'_k$	Change constraint of	Change constraint of	No impact Change constraint of	No impact Delete relation	No impact

	property of R_i	property of R_i	property of R_i	(Delete R_i & Delete relation)	Delete relation
--	-------------------	-------------------	-------------------	----------------------------------	-----------------

The following is a change propagation example.

Change Propagation Example

In the following there are two requirements for the course management system:

R61: The system shall allow lecturers to specify enrolment policies based on grade, first-come first-serve (fcfs), and department.

R62: The system shall allow lecturers to specify enrolment policies based on grade.

where R61 contains R62.

For the course management system the stakeholder needs a change. Specifying enrolment policies based on grade is not needed any more. One of the properties given in requirement R61 is allowing lecturers specifying enrolment policies based on grade. Therefore, we propose the change 'Delete property of Requirement' for R61.

Proposed Change: Delete Property of Requirement R61

Description of Change: Specifying enrolment policies based on grade is not needed any more.

The proposed change is propagated from R61 to R62 through the *contains* relation in the following:

Propagation from R61 to R62: According to Table 5.2 the alternatives to propagate the proposed change 'Delete Property of Requirement R61' to requirement R62 are (No impact | Delete Requirement R62 | Delete Property of Requirement R62).

The property to be deleted from requirement R61 is specifying enrolment policies based on grade. It should also be deleted from requirement R62. Since this property is the only property given in requirement R62, we choose the change 'Delete Requirement R62' among the change alternatives.

The following is the derivation of change alternatives for change propagation where $R_i \xrightarrow{-pt} R_i^l$ and R_i contains R_k .

Change Alternatives:

Change alternatives for R_k where $(R_i \xrightarrow{-pt} R_i^l)$ and $(R_i \text{ contains } R_k)$
 $=$ No impact | Delete R_k | Delete Property of R_k

Derivation:

Let R_i and R_k be requirements. P_i and P_k are formulas for R_i and R_k .

= {By using formalization of the contains relation}

R_i contains R_k iff P_i is derived from P_k as follows:

$$P_i = P_k \wedge P^l$$

where $P_i = \forall x((p_1 \dots p_n) \wedge (q_1 \dots q_m))$; $m, n \geq 1$ and P^l denotes properties that are not captured in P_k

= {By using formalization of the change type}

$R_i \xrightarrow{-pt} R_k^l$ iff P_i^l is derived from P_k as follows:

$$P_i^l = \forall x(p_1 \dots p_n); n \geq 1$$

where $\forall x(q_1 \dots q_m)$ denotes properties that are captured in pt.

= {By using the formalization of domain changes}

Properties $\forall x(q_1 \dots q_m)$ that are captured in pt should be deleted from the requirements model RM.

= {By using formalization of the contains relation}

There are three alternatives for P_k and impact on R_k

(i) $P_k = \forall x(z_1 \dots z_t); z \geq 1, \{z_1, \dots, z_t\} \subseteq \{p_1, \dots, p_n\}$ then No Impact

(ii) $P_k = \forall x(q_1 \dots q_m); m \geq 1$ then $\forall x(q_1 \dots q_m)$ should also be deleted. It means Delete R_k & Delete Relation

(iii) $P_k = \forall x((z_1 \dots z_t) \wedge (q_1 \dots q_m)); t, m \geq 1$ then $\forall x(q_1 \dots q_m)$ should also be deleted. It means Delete Property of R_k

All change alternatives given in Table 5.2 are derived from the semantics of change types, requirements relations and rationale of changes as shown above. Change propagation is implemented in a rule based form in TRIC (see Section 5.6). Not all derivations are given due to their size.

Proposed changes and propagated proposed changes may cause a conflict. In the following we explain how conflicts between proposed changes are identified.

Change Consistency Checking. This is the activity of identifying the proposed changes whose existence may cause a contradiction. Stakeholders may require changes that contradict with each other or the requirements engineer may propagate multiple changes to the same requirement in which the propagations cause a contradiction. Table 5.3 gives the contradicting changes based on semantics of domain changes and change types. The rows

and columns of the table are change types. Two changes for the same requirement might cause a contradiction (cells marked as *maybe* in Table 5.3) and these changes should be inspected, or there is an ensured contradiction (cells marked as *yes*) caused by these changes. Cells in Table 5.3 are marked as *no* if there is no contradiction caused by these changes.

Table 5.3 Contradicting Changes based on Semantics of Domain Changes and Change Types

Change Type	Delete R	$+pt$ $R \mapsto R'$	$-pt$ $R \mapsto R'$	$pt \mapsto pt^l$ $R \mapsto R'$	$+ct$ $R \mapsto R'$	$-ct$ $R \mapsto R'$	$ct \mapsto ct^l$ $R \mapsto R'$	No impact
Delete R	no	yes	no	yes	yes	no	yes	no
$+pt$ $R \mapsto R'$	yes	no	no	no	no	no	no	no
$-pt$ $R \mapsto R'$	no	no	no	maybe	maybe	no	maybe	no
$pt \mapsto pt^l$ $R \mapsto R'$	yes	no	maybe	maybe	maybe	maybe	maybe	no
$+ct$ $R \mapsto R'$	yes	no	maybe	maybe	no	maybe	maybe	no
$-ct$ $R \mapsto R'$	no	no	no	maybe	maybe	no	maybe	no
$ct \mapsto ct^l$ $R \mapsto R'$	yes	no	maybe	maybe	maybe	maybe	maybe	no
No impact	no	no	no	no	no	no	no	no

The following is an ensured inconsistency example.

Ensured Inconsistency Example

The following is one of the requirements for the course management system:

R7: The system shall provide a messaging facility.

There are two changes in stakeholders' needs for requirement R7. The first change is that there is no need for a messaging facility any more. The second one is that sms messaging should be provided. The followings are two proposed changes for requirement R7 based on the changes in the stakeholders' needs.

Proposed Change 1: Delete Requirement R7

Description of Proposed Change 1: There is no need for a messaging facility any more.

Proposed Change 2: Add Constraint to Property of Requirement R7

Description of Proposed Change 2: Sms messaging should be provided.

The second change is stating sms messaging as a new constraint while the first change states messaging facility is not needed at all. Therefore, there is an ensured inconsistency for these two proposed changes (see Table 5.3).

The following is a proof of this ensured inconsistency.

Ensured Inconsistency: $(R \xrightarrow{+ct} R^l) \wedge (\text{Delete } R)$

Proof Sketch: Let $R \xrightarrow{+ct} R^l$.

= {By using the semantics of domain changes and the change type 'Add Constraint to Property of Requirement'}

$$P^l = \forall x ((p_1^l \dots p_n^l) \wedge (q_1 \dots q_m)); n \geq 1, m \geq 0 \text{ and } (P^l \rightarrow P) \text{ holds} \quad (\text{a})$$

Let Delete R.

= {By using the semantics of domain changes and the change type 'Delete Requirement'}

$$P = \forall x ((p_1 \dots p_n) \wedge (q_1 \dots q_m)) \text{ and } P \text{ does not hold for the whole model} \quad (\text{b})$$

P^l in (a) states $\forall x ((p_1^l \dots p_n^l) \wedge (q_1 \dots q_m))$ holds although $\forall x ((p_1 \dots p_n) \wedge (q_1 \dots q_m))$ does not hold because P in (b) does not hold any more. Therefore, $(R \xrightarrow{+ct} R^l)$ and $(\text{Delete } R)$ contradict one another.

The following is a possible inconsistency example.

Possible Inconsistency Example

Consider the following requirement.

R61: The system shall allow lecturers to specify enrolment policies based on grade, first-come first-serve (fcfs), and department.

There are three properties in requirement R61: (i) allow lecturers to specify enrolment policies based on grade, (ii) allow lecturers to specify enrolment policies based on first-come first-serve (fcfs), and (iii) allow lecturers to specify enrolment policies based on department.

There are two changes in stakeholders' needs for requirement R61. The first change is that there is no need of specifying enrolment policies based on grade any more. The second one is that lecturers should be allowed to specify enrolment policies based on department only which they are affiliated with. The following are two proposed changes for requirement R61 based on the changes in stakeholders' needs.

Proposed Change 1: Delete Property of Requirement R61

Description of Proposed Change 1: There is no need of specifying enrolment policies based on grade any more.

Proposed Change 2: Add Constraint to Property of Requirement R61

Description of Proposed Change 2: Lecturers should be allowed to specify enrolment policies based on department only which they are affiliated with.

The first change states specifying enrolment policies based on grade is not needed any more. The second change states a constraint about departments for enrolment policies. There is a need of checking if changes are referring to the same property or not. Since two changes refer to different properties, there is no inconsistency.

The following is a proof of this possible inconsistency.

Possible Inconsistency: $(R \xrightarrow{-pt} R') \wedge (R \xrightarrow{+ct} R')$

Proof Sketch: Let $R \xrightarrow{-pt} R'$.

= {By using the semantics of domain changes and the change type ‘Delete Property of Requirement’}

$P^l = \forall x ((p_1 \dots p_n)); m, n \geq 1 \text{ and } \forall x (q_1 \dots q_m) \text{ does not hold any more}$ (a)

Let $R \xrightarrow{+ct} R'$.

= {By using the semantics of domain changes and the change type ‘Add Constraint to Property of Requirement’}

There are two alternatives for applying the change type ‘Add Constraint to Property of Requirement’ with the change type ‘Delete Property of Requirement’

$P^{ll} = \forall x (p_1^l \dots p_k^l \dots p_n)$ where $p_1^l, p_2^l, \dots, p_{k-1}^l, p_k^l$ are disjunction of literals

such that $p_u^l \rightarrow p_u$ for all $u \in 1..k$ and $k \leq n$ (b)

$P^{ll} = \forall x ((p_1 \dots p_n) \wedge (q_1^l \dots q_k^l \dots q_m))$ where $q_1^l, q_2^l, \dots, q_{k-1}^l, q_k^l$ are disjunction of literals

such that $q_u^l \rightarrow q_u$ for all $u \in 1..k$ and $k \leq m$ (c)

P^l in (a) and P^{ll} in (b) do not have any contradiction. The change type '*Add Constraint to Property of Requirement*' can be applied to requirement R with the change type '*Delete Property of Requirement*' if these two changes are applied to different properties in requirement R.

P^l in (a) and P^{ll} in (c) have a contradiction since the change type '*Add Constraint to Property of Requirement*' is applied to $\forall x(q_1 \dots q_m)$ which are not valid anymore (see (a)). Therefore, $(R \xrightarrow{-pt} R^l) \wedge (R \xrightarrow{+ct} R^{ll})$ may contradict one another.

Table 5.3 is implemented in a rule based form in TRIC. The consistency rules are checked for proposed and propagated changes (see Section 5.6).

5.5 Discussion on the Approach

The formalization of changes relies on FOL and therefore the limitations discussed in Chapter 4 are also valid here.

As we stated in Chapter 4, the requirements engineer does not need to know the details of the formalization since he/she can be guided by tutorials [94] that provide an informal explanation of the relations. Similar to tutorials for requirements relations, tutorials can be provided for the interpretation of informal change request based on our formal change classification. The requirements engineer receives the change request from the stakeholder who might be a user of the system, system developer or the project manager. Then, he/she interprets the informal change request based on the tutorial in order to propose and propagate changes over the requirements model.

Our approach has limitations for some change types and relation types. Change alternatives in Table 5.2 are used only if there is any requirement related to the changed requirement. For instance, adding a new requirement (*Add R_x*) has no impact on other requirements in the requirements models according to Table 5.2. The requirements engineer has to determine relations for the added requirement and find if there is any impact on other requirements. Also, there may be relations that are missed by the requirements engineer during modeling but appear later (see Chapter 6).

There might be multiple relations between two requirements. The priority is given to the intensionally defined relations for propagation of changes through multiple relations. For instance, we stated that the *refines* and *contains* relations imply the *requires* relation. Since *refines* and *contains* are given in intensional terms, our approach uses *refines* and *contains* to determine change alternatives.

In the implementation of change propagation and change consistency checking, change impact alternatives in Table 5.2 and contradicting changes in Table 5.3 are hard-coded. When there is a new relation and/or change type, additional manual proofs have to be implemented in the current tool support.

5.6 Tool Support

In Chapter 4, we described the Tool for Requirements Inferencing and Consistency Checking (TRIC). We extended TRIC with features for change impact analysis in requirements [235]. In this section, we give the details of the extension. In Section 5.6.1, we depict the usage of the tool in the context of a requirements modeling process. Section 5.6.2 gives the architecture of the tool. Section 5.6.3 describes the main features of the tool with some screenshots.

5.6.1 The Modeling Process

We depict the usage of the tool in a requirements modeling process with change propagation and change consistency checking. This process is based on an analysis of activities during change impact analysis. Figure 5.6 gives a UML activity diagram of the process.

The process consists of the following activities.

Modeling Requirements. This activity takes the requirements document as input and produces the requirements model which is an instance of the requirements metamodel.

The modeling process is forked into three activities: *proposing change*, *propagating change* and *checking change consistency*.

Proposing Change. This activity takes the requirements model as input and produces the proposed changes in the requirements model as output. The requirements engineer proposes changes based on the interpretation of the changes in stakeholder's needs. The activity denotes proposing a single change in the model. The modeling process is iterative and the requirements engineer may introduce multiple changes consecutively without propagating the proposed changes.

Propagating Change. The activity takes the requirements model with proposed changes as input and produces the propagated changes in the requirements model as output. The activity is semi-automatic. Propagation alternatives described in Table 5.2 are applied. The requirements engineer has to select one of the propagation alternatives proposed by the tool. The activity denotes one step propagation of a single change in the model. The modeling

process is iterative and the requirements engineer may propagate multiple changes multiple times consecutively.

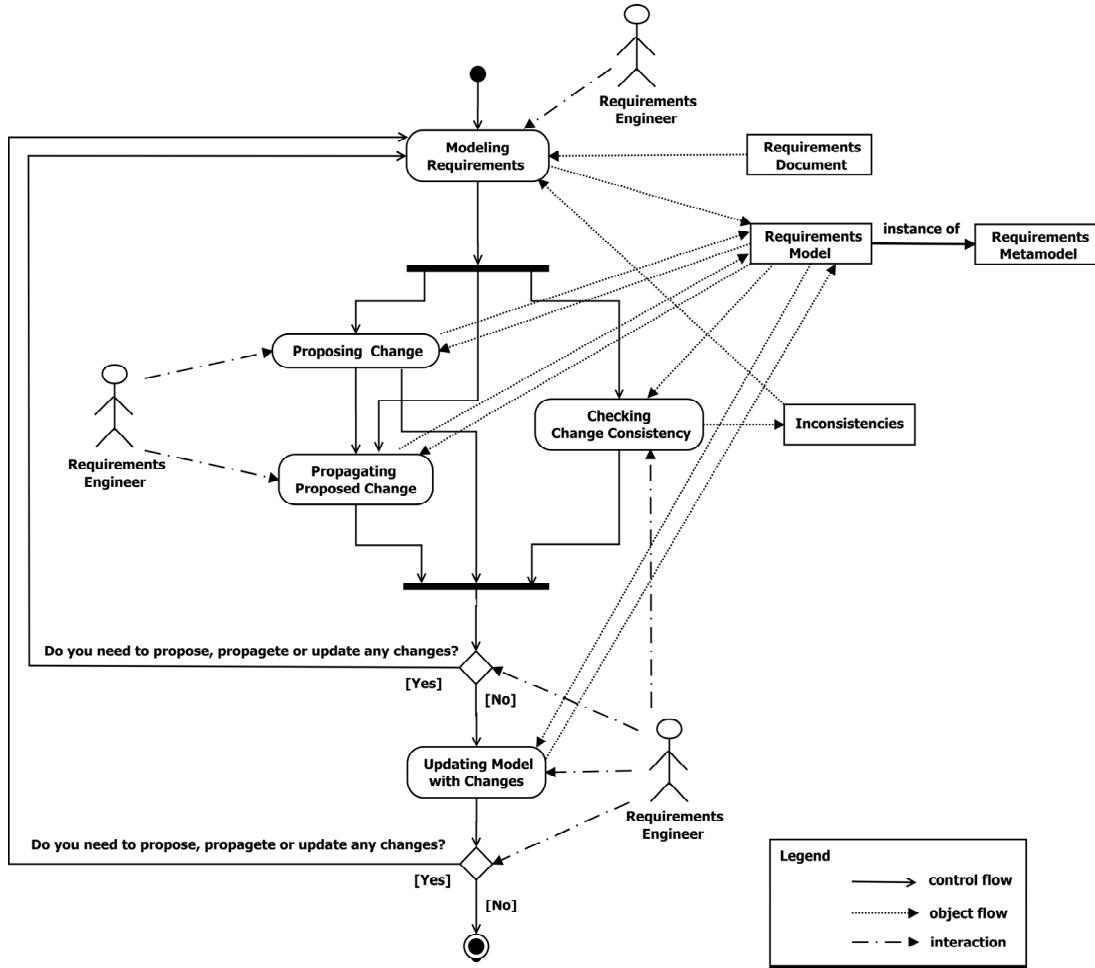


Figure 5.6 Requirements Modeling Process with Change Propagation and Change Consistency Checking

Checking Change Consistency. The activity takes the requirements model including the proposed changes and gives inconsistencies between proposed changes as output.

If there is no need to propose, propagate or update any changes further, the requirements engineer starts updating the requirements model according to proposed changes.

Updating Model with Changes. This activity takes the requirements model with proposed changes as input and produces the updated requirements model as output. The activity is manual. The requirements engineer changes requirements according to proposed changes.

Iterating. The process given in Figure 5.6 is iterative: the requirements engineer may return to the modeling activity in order to propose/propagate changes and/or update changes. If there is no need to update the model, the process is terminated.

5.6.2 Tool Architecture

The tool is composed of three layers as already given in Chapter 4: a) the *User Interface (UI) layer*, b) the *Application Layer*, and c) the *Data Layer*. Figure 5.7 gives the extended version of the layered architecture. We extended some of the existing components (Consistency Checking Engine, Visualization Engine, and Modeling Environment) in the architecture of TRIC and added some new components (Change Propagation Engine and XML file) for change impact analysis features. In this section, we explain only the extended and added components (colored gray in Figure 5.7).

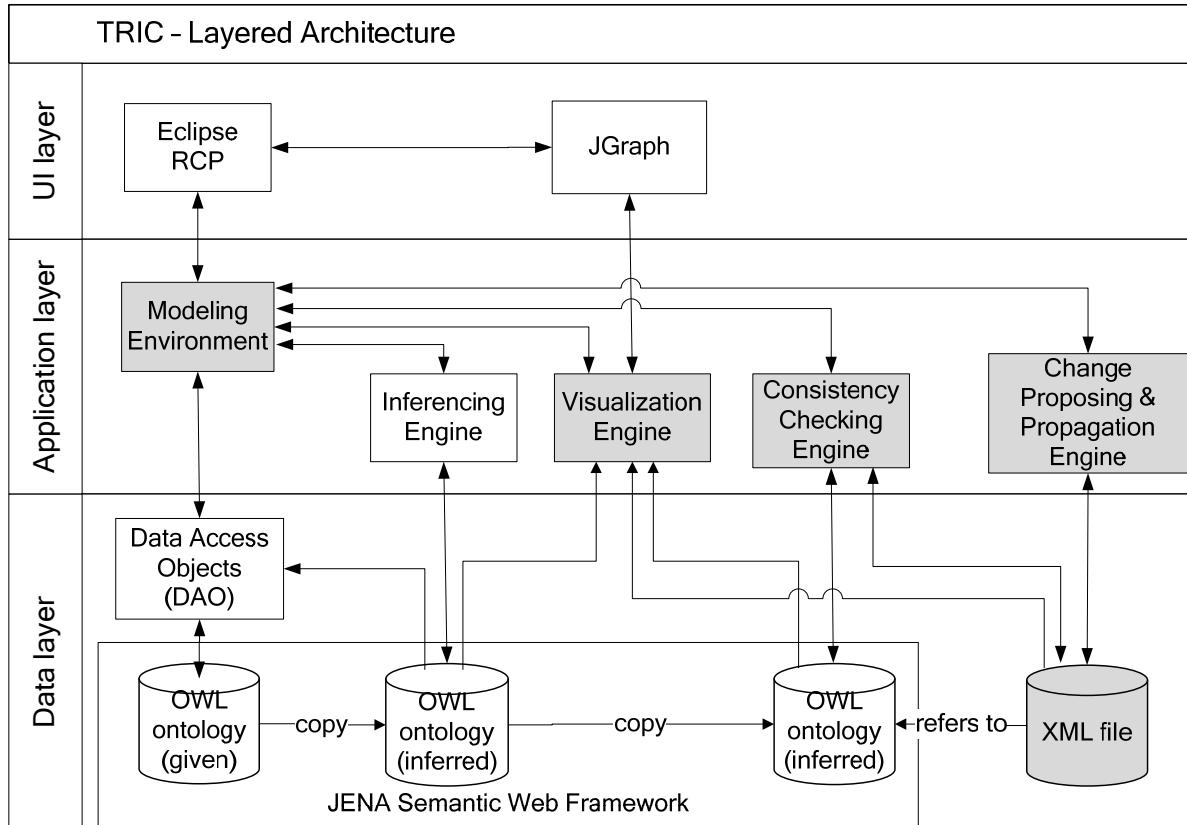


Figure 5.7 Layered Architecture of the Tool

Change Proposing & Propagation Engine. It supports the *proposing change* and *propagating change* activities. The engine asks the requirements engineer to give the relevant change type for the selected requirement. Change propagation is done by the engine based on the semantics of the proposed change and the requirements relations.

Consistency Checking Engine. This component allows checking consistency of requirements relations based on semantics of relations (see Chapter 4). It is extended to support the *checking change consistency* activity.

Visualization Engine. It accesses the Data layer in order to get requirements and relations to be visualized in diagrams (see Chapter 4). In addition to that, the visualization engine is used to show the results of change impact analysis such as the propagation path of a change.

Modeling Environment. This component allows the creation, storage, and retrieval of requirements models and bridging the User Interface layer with the Data layer (see Chapter 4). It is extended for the *updating model with changes* activity in Figure 5.6.

XML File. The proposed and propagated changes are stored in XML format. We split up the requirements model in OWL ontology and proposed & propagated changes in XML file for separation of concerns. The XML file always refers to the relevant OWL ontology to relate the changes with the requirements model. Therefore, we can have different versions of proposed & propagated changes for the same requirements model.

5.6.3 Tool Features

We describe the most important features of the tool: *proposing changes*, *propagating changes*, *displaying inconsistencies in proposed changes*, *implementing proposed changes in the requirements model*, and *predicting impact of proposed changes*.

Proposing changes. Figure 5.8 gives the GUI for proposing changes. The left-hand side of the window lists the requirements in the model. The right-hand side of the window shows the details of the selected requirement (R7). The *Propose Change* window opened by right clicking on the selected requirement (R7) is used to propose a change with a type and description of the change.

After proposing a change, the tool lists the requirements related to the changed requirement. These requirements are candidate impacted (CI) requirements in which the requirements engineer can propagate the change (see Figure 5.9).

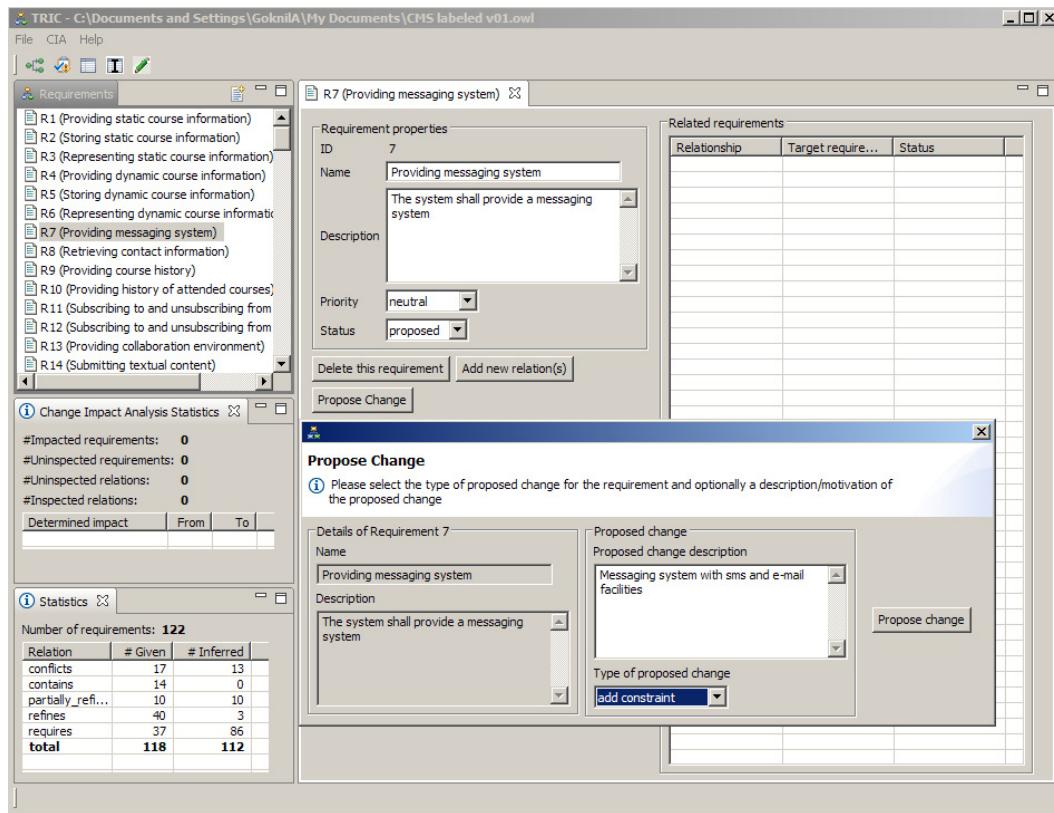


Figure 5.8 GUI for Proposing Changes

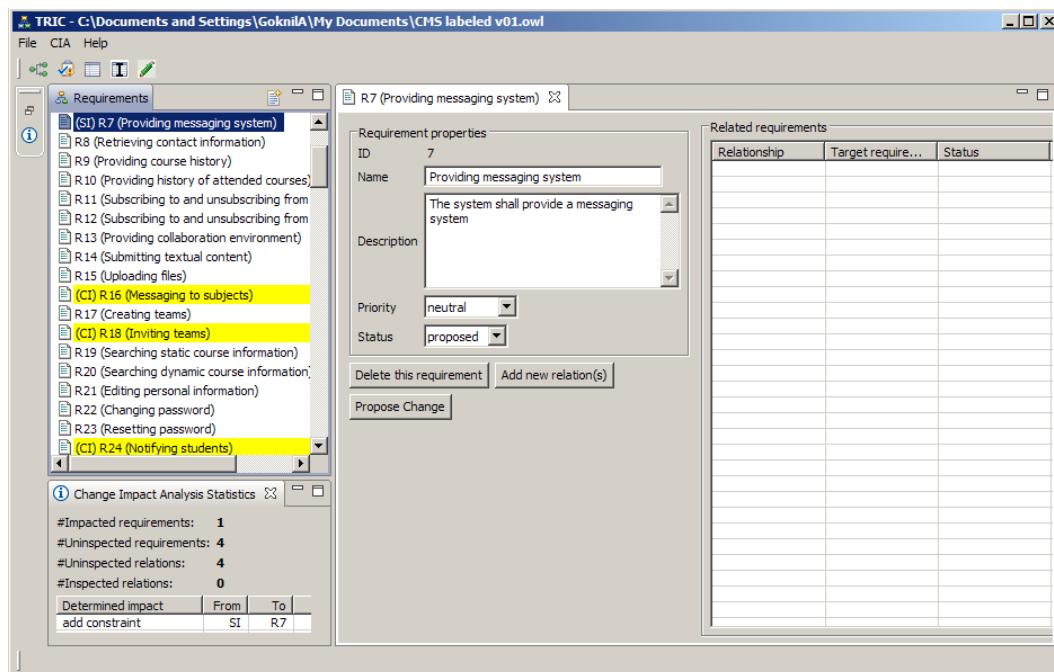


Figure 5.9 Output of the Proposing Change Activity

The left-hand side of the window lists the requirements in the model with the proposed change requirement (R7) tagged as Starting Impacted (SI) and related requirements (R16, R18, and R24) tagged as Candidate Impacted (CI). The requirements engineer can select the candidate impacted requirements (R16, R18, and R24) to propagate the proposed change to them. At the bottom of the left-hand side of the window, the numbers about change impact analysis such as numbers of impacted requirements and uninspected requirements are listed.

Propagating changes. Figure 5.10 gives the GUI for propagating proposed changes which supports the *propagating change* activity in Figure 5.6.

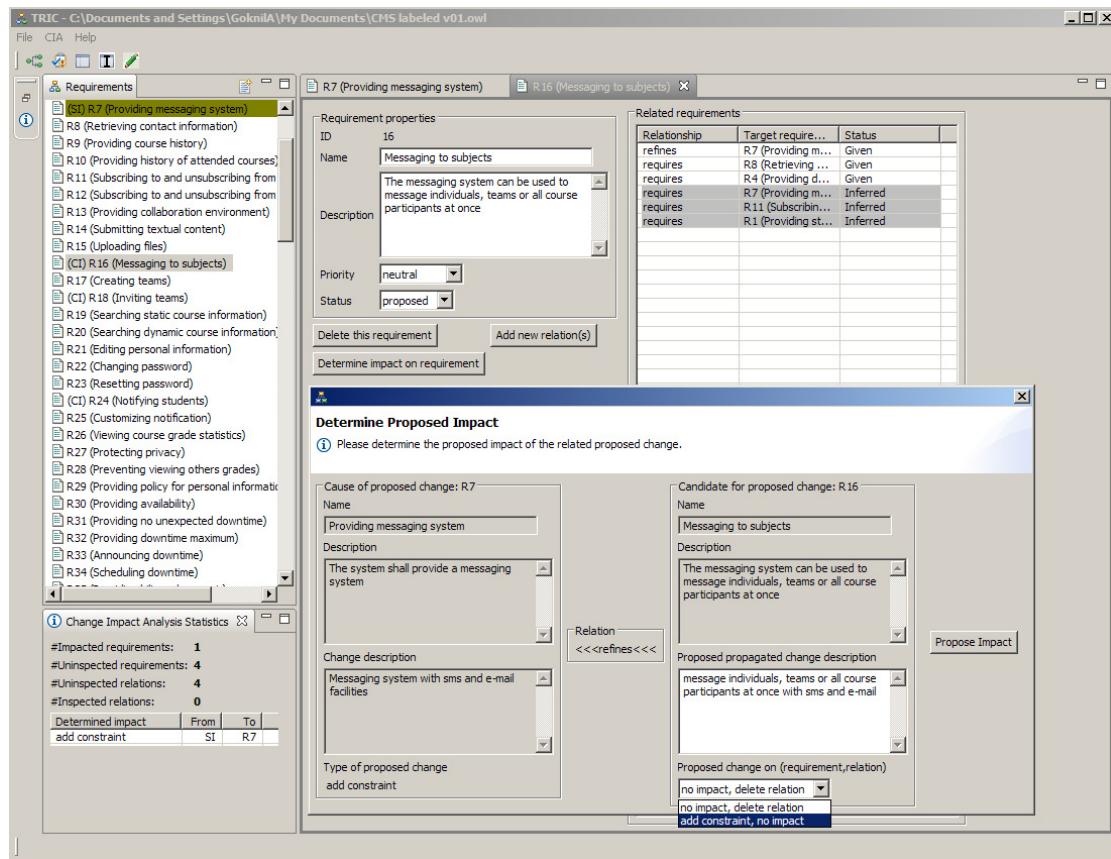


Figure 5.10 GUI for Propagating Proposed Changes

The *Determine Proposed Impact* window is opened by clicking on one of the candidate impacted requirements (R16). The tool asks the type of the proposed change for candidate impacted requirement with a change description.

We support a matrix view in order to represent and propagate changes. Such a view is also available in commercial requirements management tools, such as RequisitePro in order to determine the impacted requirements. Figure 5.11 illustrates the matrix view.

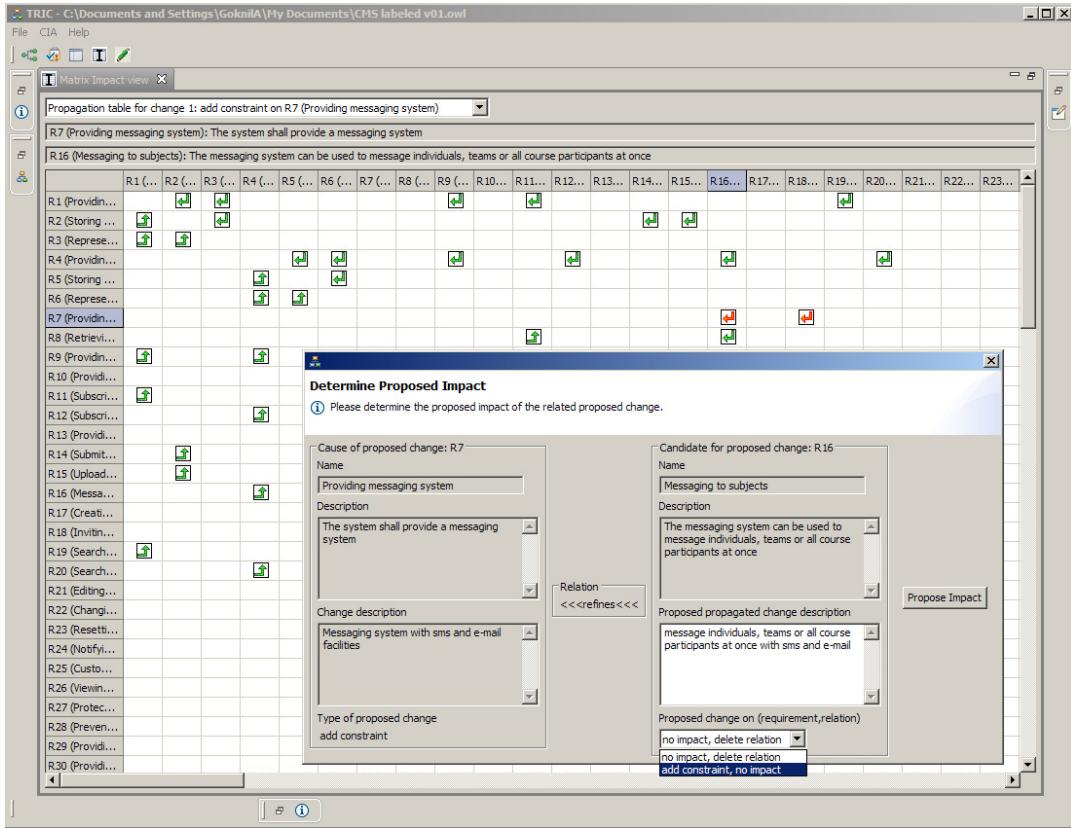


Figure 5.11 Matrix View for Propagating Proposed Changes

The arrows (green and red) with direction in the cells denote the existence of requirements relations with their directions. In addition to that, the red arrows indicate the candidate impacted requirements for the proposed change in the selected requirement (R7) listed at the top of the window. By clicking the red arrows, the tool provides the Determine Proposed Impact window, which is similar to the window in Figure 5.10. Since there can be multiple proposed changes in the requirements model, the tool has a different matrix view for each proposed change.

The GUI for propagating proposed changes in Figure 5.10 and the Impact Matrix view in Figure 5.11 do not allow analysis of multiple proposed changes simultaneously. To support simultaneous analysis of multiple impact propagations, tool support for building decision trees is provided. Figure 5.12 shows the interactive decision tree builder for propagating changes.

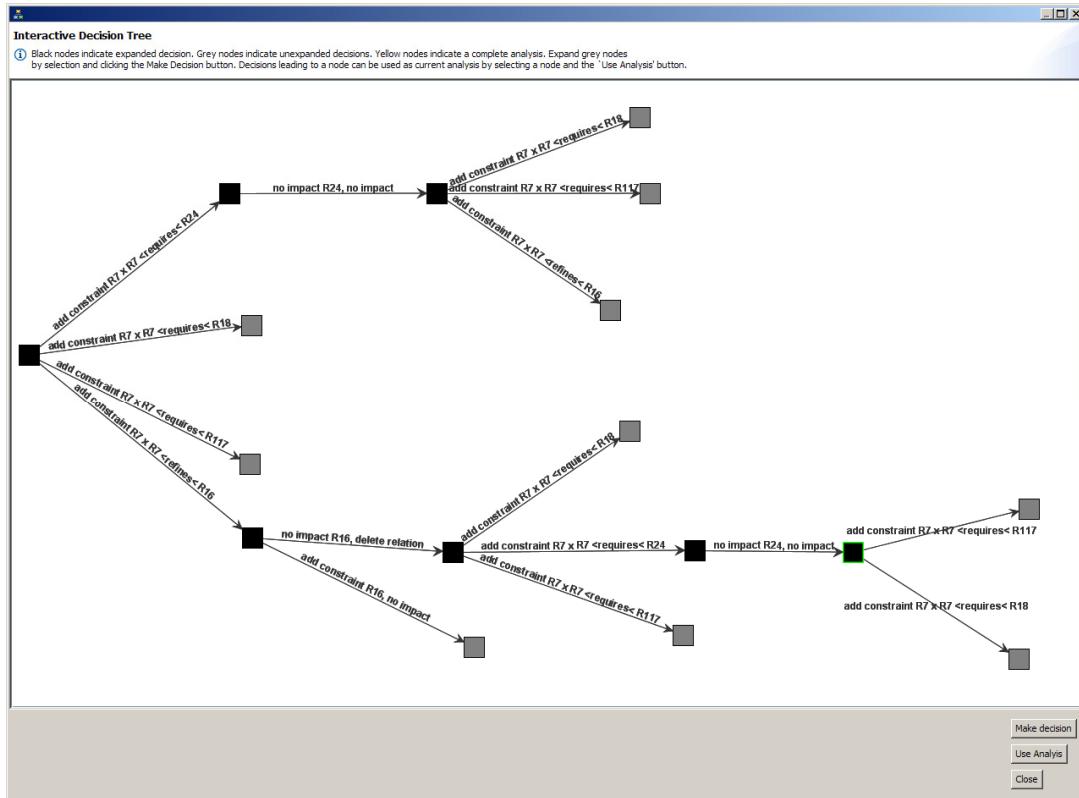


Figure 5.12 Interactive Decision Tree Builder for Propagating Proposed Changes

Each arrow in Figure 5.12 indicates a decision captured by the target node of the arrow. The decision tree can be expanded by making decisions (the *Make Decision* button in Figure 5.12). Once the analysis of the interactive decision tree is concluded, the requirements engineer can select a node and apply decisions captured by the path from the tree root to the selected node (the *Use Analysis* button in Figure 5.12).

Displaying inconsistencies in proposed changes. Figure 5.13 gives the screenshot of the tool for the output of the *checking change consistency* activity. The window in Figure 5.13 has a list view including three columns. The first column of the list view gives the requirements that have contradicting proposed changes. The second column shows if the inconsistency is *ensured* or *possible*. The third column lists proposed changes that cause the inconsistency for the given requirement.

The tool provides an explanation of contradicting proposed changes, for example, the contradicting proposed changes “Add Constraint to Property of Requirement” and “Delete Requirement” for requirement R16 (see Figure 5.14).

The screenshot shows a window titled 'Impact Inconsistencies'. A message at the top says: 'The following (possible) inconsistencies are found. Right-click on a requirement to inspect.' Below is a table:

Requirement	Degree	Contradicting impacts
R16	ensured	Impact 1: add constraint and impact 2: delete requirement
R36	ensured	Impact 1: delete requirement and Impact 2: add property
R40	possible	Impact 1: delete property and Impact 2: add constraint
R44	possible	Impact 1: change constraint and Impact 2: delete property

Figure 5.13 Output of the Checking Change Consistency Activity

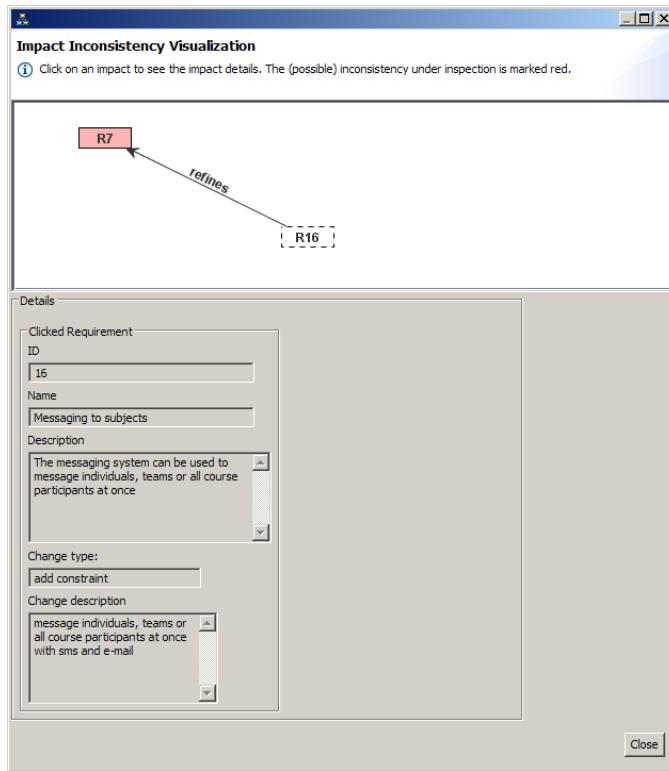


Figure 5.14 Explanation of the Proposed Change of R16 Causing the Inconsistency

Figure 5.14 shows the propagation path for the proposed change “Add Constraint to Property of Requirement” in R16. R7 is the requirement where first the change is proposed and this change is propagated to R16 as “Add Constraint to Property of Requirement” by using the *refines* relation between R7 and R16.

Implementing proposed changes in the requirements model. The tool allows 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 (see Figure 5.15). Then, propagated proposed changes are implemented (see Figure 5.16).

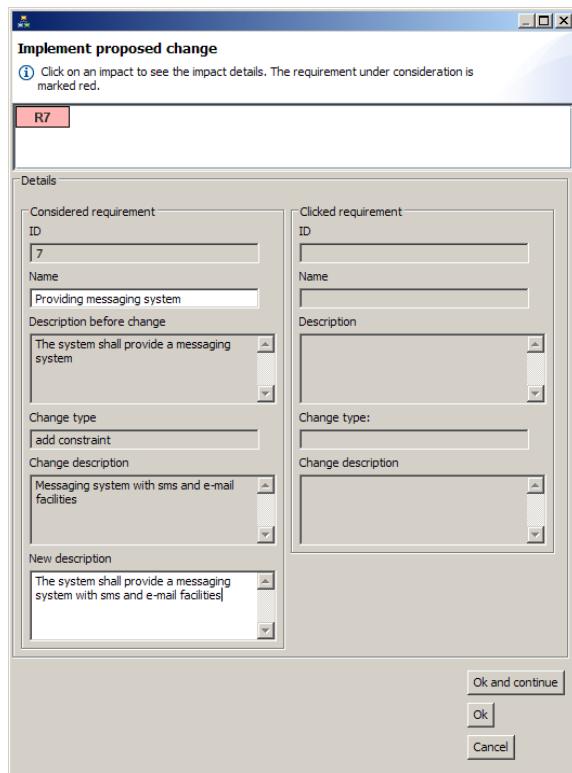


Figure 5.15 GUI for Implementing Proposed Changes

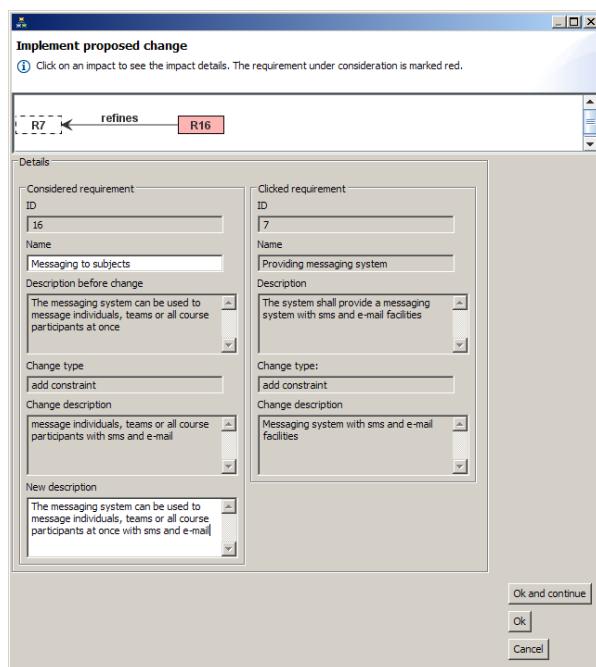


Figure 5.16 GUI for Implementing Propagated Proposed Changes

Predicting impact of proposed changes. 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 5.17 gives the output of the impact prediction for the proposed change “Add Constraint to Property of Requirement” in R7.

The window in Figure 5.17 has a list view including three columns. The first column of the list view gives requirements in the model. The second column shows if the requirement is impacted (*no*, *yes*, or *maybe*). The third column lists the impact type if there is any impact. For instance, in Figure 5.17, R16 might be impacted by the proposed change “Add Constraint to Property of Requirement” in R7 and the type of change for the possible impact in R16 is “Add Constraint to Property of Requirement”.

The tool also provides the propagation paths for the impacts listed in Figure 5.17. Figure 5.18 gives the propagation paths for the impact in R16. The first part of the window in Figure 5.18 gives the types of impact for R16. There are two change types to be propagated for R16: “Add Constraint to Property of Requirement” or “No Impact”. The second part of the window shows the requirements in the propagation path.

Requirement	Impacted?	Impact Types
R1	no	no impact
R2	no	no impact
R3	no	no impact
R4	no	no impact
R5	no	no impact
R6	no	no impact
R7	yes	add constraint
R8	no	no impact
R9	no	no impact
R10	no	no impact
R11	no	no impact
R12	no	no impact
R13	no	no impact
R14	no	no impact
R15	no	no impact
R16	maybe	add constraint,no impact
R17	no	no impact
R18	no	no impact
R19	no	no impact
R20	no	no impact
R21	no	no impact
R22	no	no impact
R23	no	no impact
R24	no	no impact
R25	no	no impact
R26	no	no impact
R27	no	no impact
R28	no	no impact
R29	no	no impact
R30	no	no impact
R31	no	no impact
R32	no	no impact
R33	no	no impact
R34	no	no impact
R35	no	no impact
R36	no	no impact
R37	no	no impact
R38	no	no impact
R39	no	no impact
R40	no	no impact

Figure 5.17 Output of the Impact Prediction for the Proposed Change in R7

Type of impact	Requirements captured by path	Number of paths
add constraint	R7,R16	1
no impact	R7,R16	1
no impact	R1,R2,R4,R7,R13,R15,R16,R17,R18,R97	1
no impact	R1,R2,R4,R7,R13,R14,R16,R17,R18,R97	1
no impact	R1,R2,R4,R7,R9,R13,R14,R16,R17,R18	1
no impact	R1,R2,R7,R8,R11,R13,R14,R16,R17,R18	1
no impact	R1,R2,R4,R7,R9,R13,R15,R16,R17,R18	1
no impact	R1,R2,R4,R7,R8,R11,R13,R14,R16,R17,R18	1
no impact	R1,R2,R3,R7,R8,R11,R13,R15,R16,R17,R18	1
no impact	R1,R2,R3,R4,R7,R13,R15,R16,R17,R18,R97	1
no impact	R1,R2,R3,R4,R7,R13,R14,R16,R17,R18,R97	1
no impact	R1,R2,R3,R4,R7,R9,R13,R15,R16,R17,R18	1
no impact	R1,R2,R4,R7,R11,R13,R14,R16,R17,R18,R60,R97,R100,R102	1
no impact	R1,R2,R4,R7,R11,R13,R15,R16,R17,R18,R60,R97,R100,R102	1
no impact	R1,R2,R7,R8,R11,R13,R14,R16,R17,R18,R60,R97,R100,R102	1
no impact	R4,R6,R7,R13,R15,R16,R17,R18,R26,R53,R68,R84,R85,R90	1
no impact	R1,R2,R7,R8,R11,R13,R14,R16,R17,R18,R60,R97,R100,R102	1
no impact	R1,R2,R4,R7,R11,R13,R15,R16,R17,R18,R59,R60,R97,R100	1
no impact	R1,R2,R3,R7,R8,R11,R13,R15,R16,R17,R18,R59,R60,R97,R100	1
no impact	R1,R2,R7,R8,R11,R13,R14,R16,R17,R18,R59,R60,R97,R100	1
no impact	R1,R2,R7,R8,R11,R13,R15,R16,R17,R18,R59,R60,R97,R100	1
no impact	R1,R2,R7,R8,R11,R13,R14,R16,R17,R18,R59,R60,R97,R100	1
no impact	R1,R2,R3,R7,R8,R11,R13,R14,R16,R17,R18,R60,R97,R100,R102	1
no impact	R1,R2,R4,R7,R11,R13,R14,R16,R17,R18,R60,R97,R100,R102	1
no impact	R1,R2,R3,R4,R7,R11,R13,R14,R16,R17,R18,R60,R97,R100	1
no impact	R1,R2,R3,R4,R7,R11,R13,R14,R16,R17,R18,R60,R97,R100	1
no impact	R1,R2,R3,R7,R8,R11,R13,R15,R16,R17,R18,R59,R60,R97,R100	1
no impact	R1,R2,R4,R7,R11,R13,R15,R16,R17,R18,R59,R60,R97,R100,R104	1
no impact	R1,R2,R4,R6,R7,R8,R9,R13,R14,R16,R17,R18,R26,R27,R28,R29	1
no impact	R1,R2,R4,R7,R11,R13,R14,R16,R17,R18,R59,R60,R97,R100,R104	1
no impact	R1,R2,R4,R6,R7,R8,R9,R13,R15,R16,R17,R18,R26,R27,R28,R29	1
no impact	R1,R2,R7,R8,R11,R13,R15,R16,R17,R18,R59,R60,R61,R97,R100,R104	1
no impact	R1,R2,R4,R7,R8,R9,R13,R15,R16,R17,R18,R59,R60,R97,R100	1
no impact	R1,R2,R7,R8,R11,R13,R14,R16,R17,R18,R59,R60,R61,R97,R100,R104	1
no impact	R1,R2,R4,R6,R7,R8,R13,R14,R16,R17,R18,R26,R27,R28,R29,R97	1

Figure 5.18 Output of the Prediction Investigation for the Proposed Change in R16

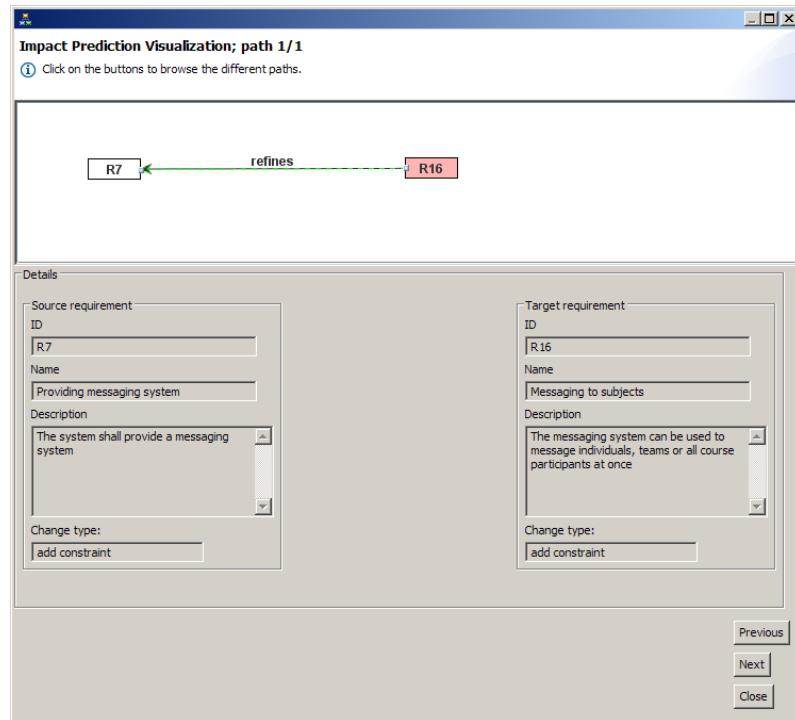


Figure 5.19 GUI for the Visualization of the Propagation Paths in Impact Prediction

Change propagation paths generated by the impact prediction are visualized by the tool. For instance, the first row in Figure 5.18 lists the propagation path from R7 to R16 with the change “Add Constraint to Property of Requirement” for R16. The visualization of this path is in Figure 5.19.

The impact prediction option allows showing the impact of the proposed change for the whole model. It is useful for large models where the matrix view does not scale well.

5.7 Example: Course Management System

In this section, we illustrate our approach and tool support with the CMS example which we also use in Chapter 4. All requirements used in this chapter can be found in Appendix B. We performed two iterations of the modeling process for the example.

- In the first iteration, we modeled the textual requirements and their relations. This is the modeling activity given in Chapter 4. Then, we proposed changes and propagated changes with the help of the tool.
- In the second iteration, we updated the model in order to correct the inconsistent proposed changes. We implemented the approved changes in the requirements model.

The example illustrates potential benefits and limitations of the approach for larger case studies. Change impact alternatives, elimination of false positive impacts in change propagation and consistency checking of changes are the potential benefits of the approach illustrated in this section. The main limitation is that the approach heavily depends on the requirements relations. False requirements relations assigned by the requirements engineer cause wrong propagation alternatives. Section 5.7.1 gives some proposed and propagated changes in the example. In Section 5.7.2, we show some inconsistent proposed changes detected in the example requirements model.

5.7.1 Proposing and Propagating Requirements Changes

Consider the following change to R7.

R7: The system shall provide a messaging facility.

Proposed Change is the following.

Change: Add constraint to property of requirement

Description of Change: Messaging facility should also contains sms and e-mail features

R7 states a messaging facility where sms and e-mail features are introduced as types of messages for messaging facility in the description of change. Since these features are constraints for the property messaging facility, the type of change is ‘Add constraint to Property of Requirement’. Then, the proposed change is propagated to requirements related to R7. Figure 5.20 gives requirements related to R7 with depth 2 (dotted arrows are inferred relations).

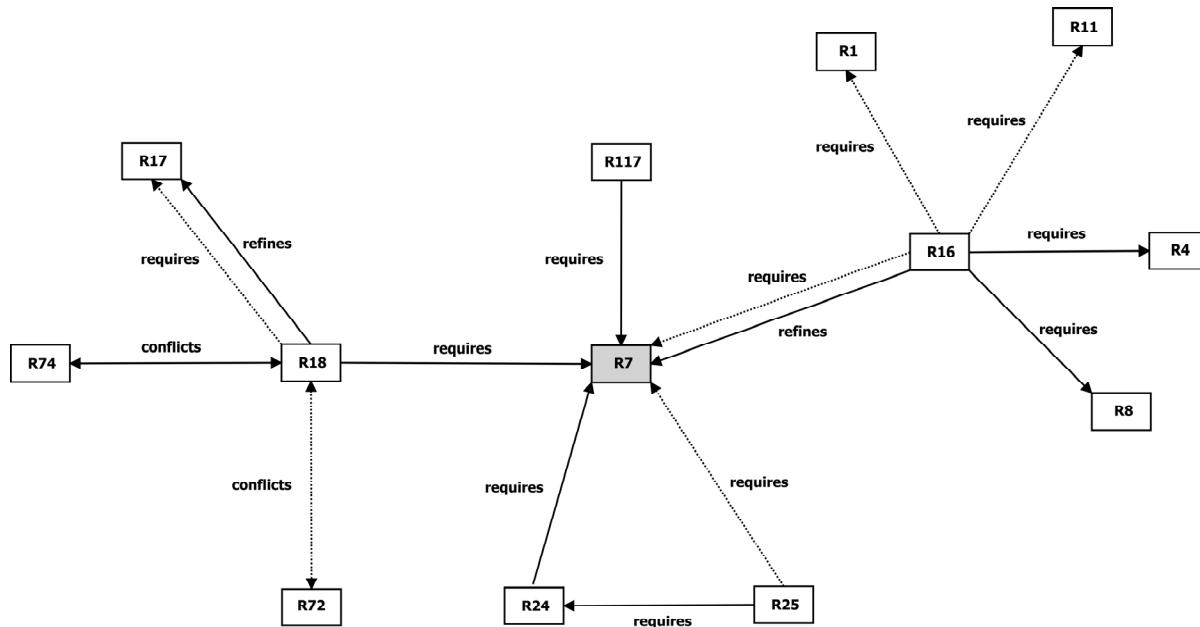


Figure 5.20 Requirements Related to R7 with Depth 2

R16: The system shall allow messages to be sent to individuals, teams, or all course participants at once.

R18: Teams are created by students inviting other students in the same course using the messaging system.

R24: The system shall notify students about events (new messages posted, etc.).

R25: The system shall allow students to customize the notification behavior.

R117: The system shall allow the administration to evaluate courses through students by means of a web-survey.

According to Table 5.2 in Section 5.4, there is no impact for R18, R24, R25 and R117, which require R7, for the proposed change in R7. Then, we do not have to check R17, R74 and R72 since they are indirectly related to R7 through R18.

There are two change alternatives to propagate the proposed change from R7 to R16 via the *refines* relation: ‘Add Constraint to Property of Requirement’ or ‘Delete Relation’. The change type ‘Add Constraint to Property of Requirement’ is chosen among these two to be proposed for R16 since the constraint added to R7 is also a constraint for R16.

Proposed Change for the requirement R16 is the following.

Change: Add constraint to property of requirement

Description of Change: Messages to be sent to individuals, teams, or all course participants at once with both sms and e-mail.

The next propagation of proposed change is from R16 to related requirements. Figure 5.21 gives requirements related to R16 with depth 2 (inferred relations are not shown for simplicity).

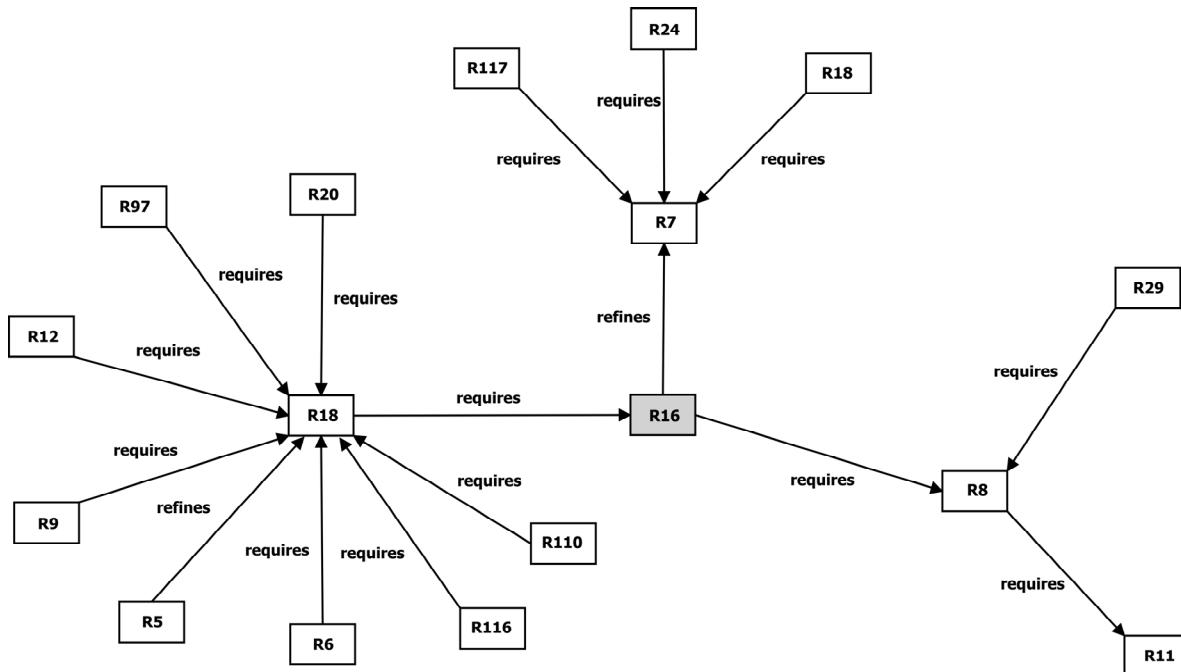


Figure 5.21 Requirements Related to R16 with Depth 2

R8: The system shall enable students to retrieve contact information of students and lecturers of subscribed courses.

According to Table 5.2, for the proposed change in R16, there is no impact for R8 and R18 which are related to R16 with the *requires* relation. Then, we do not have to check R5, R6,

R9, R12, R20, R97, R110 and R116 since they are indirectly related to R7 through R18. There are no other requirements related to R16 and the change propagation is over.

5.7.2 Checking Consistency

In this section, we discuss inconsistencies which are detected by our tool. R16 is the requirement which has the inconsistent proposed changes. They are the following:

Change 1: Add Constraint to Property of Requirement R16

Description of Change 1: Messages to be sent to individuals, teams, or all course participants at once with sms and e-mail.

Change 2: Delete Requirement R16

Description of Change 2: Messaging individuals, teams, or all course participants is not required any more.

According to Table 5.3, changes “Add Constraint to Property of Requirement” and “Delete Requirement” cause an ensured inconsistency. Since the change “Add Constraint to Property of Requirement” is a propagated proposed change, we also need to analyze change propagation path of this change. Figure 5.22 gives the propagation path of the proposed change for R16 in the inconsistency.

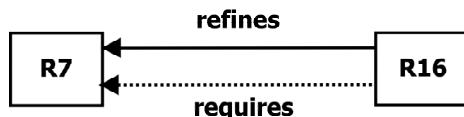


Figure 5.22 Propagation Path of the Proposed Change for R16 in the Inconsistency

According to the propagation path in Figure 5.22, the proposed change in R16 is caused by propagating the change in R7 via the *refines* relation. In order to fix the inconsistency, the requirements engineer has three options. He/she might decide that the proposed change “Delete Requirement” in R16 is not valid, or the proposed change “Add Constraint to Property of Requirement” in R7 is not valid. The third option is that the change alternative “Delete Relation” is chosen to propagate the proposed change “Add Constraint to Property of Requirement” in R7 to R16 (see Section 5.7.1). This decision has to be made as a result of negotiation between the requirements engineer and the stakeholder who has the change request.

5.8 Evaluation of the Approach

In this section we compare our approach with one of the industrial requirements management tools IBM Rational RequisitePro. As we discuss in Section 5.9, most of the approaches and tools like IBM Rational RequisitePro [119] do not focus on formal semantics of requirements relations and change types. By using formal semantics we provide a more precise change impact analysis in requirements models because we have the following features in our approach:

- change alternatives in change propagation,
- elimination of false positive impacts in change propagation,
- consistency checking of changes.

In the following we compare our approach with RequisitePro based on these features.

Change Alternatives in Change Propagation. Our approach provides a classification of changes in requirements models (see Table 5.1 in Section 5.3). The requirements engineer proposes a change with a type before implementing the change in the model. The main advantage of our approach with change types is that propagation alternatives are provided to be chosen by the requirements engineer. Change alternatives provide information to the requirements engineer about what to change in impacted requirements. Table 5.4 gives a part of change impact alternatives for our approach and RequisitePro.

Table 5.4 Part of Change Impact Alternatives for Our Approach and IBM Rational RequisitePro

Changes	Requirements Relation Types						
	Relation Types in Our Approach					Relation Type in RequisitePro	
	R _i contains R _k	R _i refines R _k	R _i partially refines R _k	R _i requires R _k	R _i conflicts R _k	R _i trace from R _k	R _i trace to R _k
R _i $\xrightarrow{+pt}$ R _i '	No impact	Add property to R _k Delete relation	Delete relation	No impact	No impact	No impact Change R _k	No impact Change R _k
R _i $\xrightarrow{-pt}$ R _i '	No impact Delete relation (Delete R _k & Delete relation)	Delete property of R _k (Delete property of R _k & Delete)	Delete property of R _k	No impact Delete relation (Delete R _k & Delete)	No impact Delete relation	No impact Change R _k	No impact Change R _k

	Delete property of R _k	relation)		relation)			
$R_k \xrightarrow{pt \mapsto pt^1} R'_k$	No impact Change property of R _k	Change property of R _k (Change property of R _k & Delete relation)	Change property in R _k (Delete R _k & Delete relation)	No impact Delete relation (Delete R _k & Delete relation)	No impact Delete relation	No impact Change R _k	No impact Change R _k

In Table 5.4 there are three change types and their propagation alternatives provided by our approach and RequisitePro. RequisitePro has only two relation types (*traceFrom* and *traceTo*) with informal definitions. As shown in Table 5.4, for each change type, RequisitePro provides two alternatives (*No impact* or *Change R_k*) since there is only one change type (*Change requirement*). In RequisitePro, the requirements engineer has to inspect the impacted requirement in order to determine the type of change without any semantic information. In our approach, the requirements engineer inspects the impacted requirement based on the change alternatives derived from the semantics of requirements relations and change types. On the other hand, the requirements engineer has to spend some effort to model requirements and to determine their relations before performing change impact analysis in our approach. Figure 5.23 gives the requirements related to R7 in the CMS requirements model with depth 2 in RequisitePro (see Figure 5.20 in Section 5.7.1 for the correspondence model in TRIC).

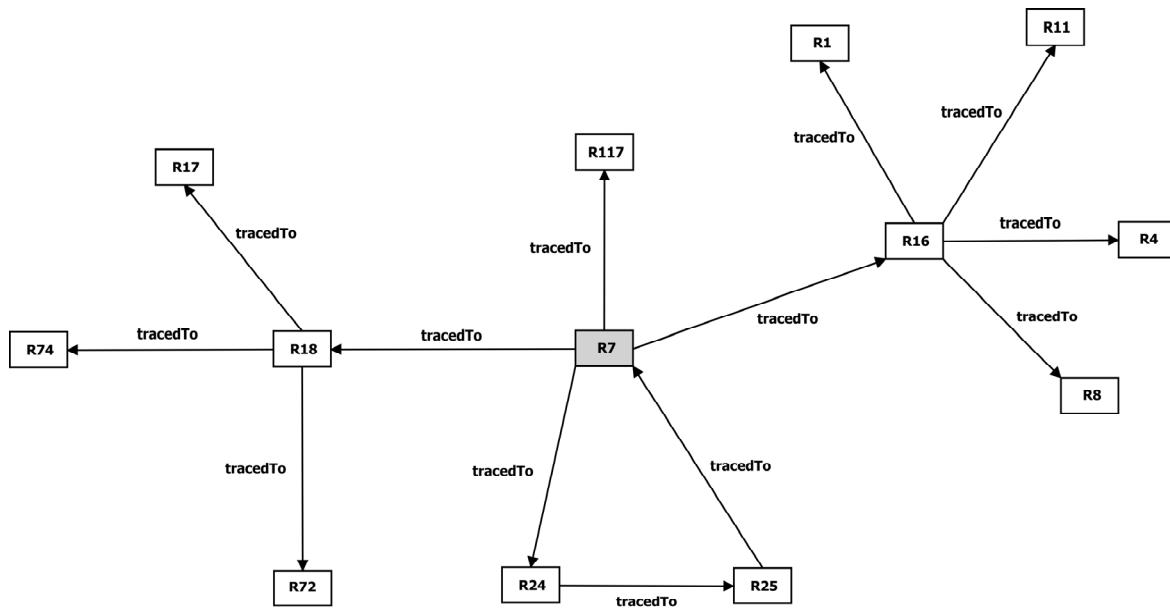


Figure 5.23 Requirements Related to R7 with Depth 2 in IBM Rational RequisitePro

Please note that RequisitePro does not provide visualization like the one in Figure 5.23. We converted the part of the matrix view of the CMS requirements model in RequisitePro to the graph visualization. Consider the following change to R7.

R7: The system shall provide a messaging facility.

Description of Change: Messaging facility should also contain sms and e-mail features

Since RequisitePro does not support proposing changes based on a change classification, the change is implemented by updating R7. Requirements relations for R7 get suspended after implementing the change in R7 in RequisitePro. Figure 5.24 shows the suspended relations in the matrix view.

R16: The system shall allow messages to be sent to individuals, teams, or all course participants at once.

R18: Teams are created by students inviting other students in the same course using the messaging system.

R24: The system shall notify students about events (new messages posted, etc.).

R25: The system shall allow students to customize the notification behavior.

R117: The system shall allow the administration to evaluate courses through students by means of a web-survey.

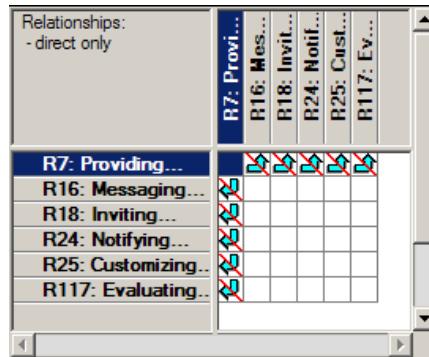


Figure 5.24 Suspended Relations for Impacted Requirements by the Change in R7

All requirements directly related to R7 with the suspended relations (R16, R18, R24, R25 and R117) are candidate impacted. The requirements engineer has to inspect the candidate impacted requirements to identify changes if there is any. When the same change is proposed with the change type ‘Add Constraint to Property of Requirement’ in our approach, ‘no impact’ is automatically identified for R18, R24, R25 and R117 (see Section

5.7.1). Our approach provides two change alternatives to propagate the proposed change from R7 to R16 via the *refines* relation: ‘Add Constraint to Property of Requirement’ or ‘Delete Relation’. The requirements engineer inspects R16 to propose a change among these two alternatives.

Elimination of False Positive Impacts in Change Propagation. Without employing any semantics information about relations and change types, all requirements directly related to the changed requirement are identified as candidate impacted. The requirements engineer has to check all these requirements manually to identify which requirements are actually not impacted (false positive impacts). For some change and relation types, our approach identifies ‘no impact’ for the related requirements. For instance, for the change in R7, all requirements (R16, R18, R24, R25 and R117) related to R7 are identified as candidate impacted by RequisitePro. All of these requirements are checked to identify the change although there is no impact for R18, R24, R25 and R117. When the change in R7 is proposed with the change type ‘Add Constraint to Property of Requirement’ in our approach, ‘no impact’ is automatically identified for R18, R24, R25 and R117 which are false positive impacts (see Section 5.7.1).

Apart from directly related requirements, there might be other candidate impacted requirements indirectly related to the changed requirement. Figure 5.25 shows some of the requirements directly/indirectly related to R7 at a distance of 1, 2, 3 and 4. Here, distance is the number of relations between two related requirements [24].

The requirements indirectly related to R7 at a distance of 2, 3 and 4 (see Figure 5.25(b), (c) and (d)) are candidate impacted to be inspected in RequisitePro. By following directly and indirectly related requirements like in Figure 5.25, the number of impacted requirements might explode at some distance [24].

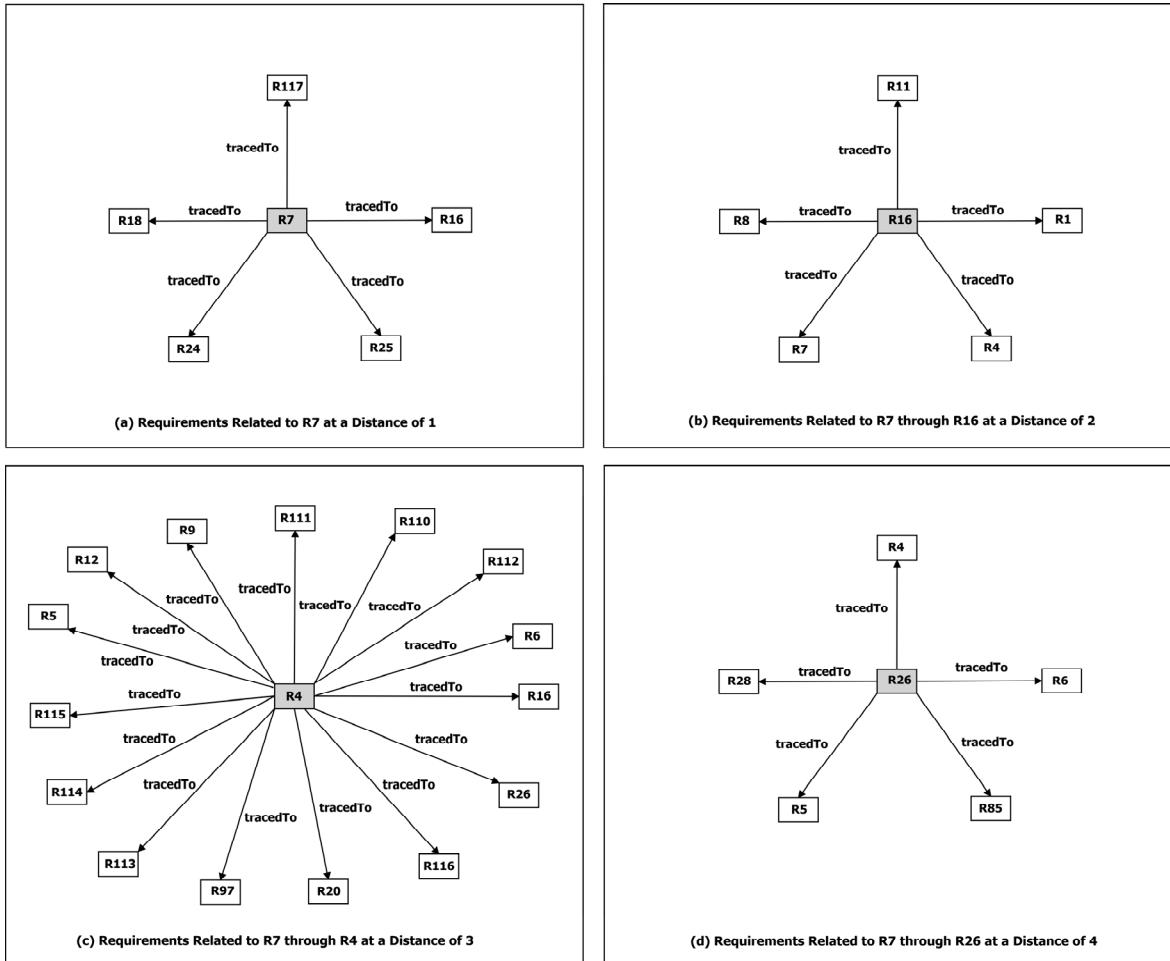


Figure 5.25 Some of the Requirements Directly/Indirectly Related to R7 in RequisitePro

Our approach provides impact prediction for a proposed change (see Section 5.6.3). The output of the impact prediction is the impacted requirements including both directly and indirectly requirements with change alternatives. For instance, for the change in R7, the output of the impact prediction is that only R16 might be impacted with the change type ‘Add Constraint to Property of Requirement’ (see Figure 5.17 in Section 5.6.3). All other impacts identified by following directly and indirectly related requirements in RequisitePro are false positives. In this way we reduce the number of elements to be inspected.

Consistency Checking of Changes. Our approach provides consistency checking of changes based on the formal semantics of requirements, relations and changes (see Section 5.7.2 for the example of consistency checking). RequisitePro does not support any consistency checking activity for requirements changes.

5.9 Related Work

We classify the related work in three categories: *Change Classification with Formal Semantics*, *Change Impact Analysis in Requirements*, and *Tool Support*.

5.9.1 Change Classification with Formal Semantics

We studied literature about requirements change classification and semantics of change types. Buckley et al. [40] propose a taxonomy of software change based on characterizing the mechanisms of change and the factors that influence these mechanisms. In this taxonomy, change type is one of the characterizing and influencing factors for mechanisms of change. Change types in [40] are defined as *structural* and *semantic* changes. Structural changes are changes that alter the structure of software. Another distinction for changes is *semantics-preserving* and *semantics-modifying* changes. This distinction is very much similar to our classification of change rationale named as *domain changes* and *refactoring*. Buckley et al., however, focus more on semantics of software components, such as type hierarchy, scoping, visibility, accessibility, and overriding relationships, rather than on changes in requirements.

Kitchenham et al. [140] propose an ontology to identify a number of factors that influence maintenance. The ontology has *Modification Activity* as an entity, specialized by *Enhancement* and *Correction* entities. In *Corrections*, a defect such a discrepancy between the required behaviour of a product/application and the observed behaviour is corrected [140]. *Enhancements* might be the changes in the implementation or they might be requirements changes which are *adding new requirements* or *changing existing requirements*. According to Kitchenham, “Add a new Requirement” and “Update an Existing Requirement” can be aquated to Swanson’s adaptive and perfective maintenance change types [237] [238] in turn. The difference with our work is that requirements change types in [140] have no formal semantics.

Aizenbud-Reshef et al. [6] present an approach to defining operational semantics for a trace in UML. The semantic property of a trace is a triplet (*event*, *condition* and *actions*). An event indicates a change. Conditions help to differentiate among events. Actions describe what should and should not be done when a specific event has occurred. There are event types (*delete events*, *update events*, and *create events*) which can be considered as change types. The main goal is to achieve automated consistency management of UML class diagrams. Therefore, it is hard to use the semantics in [6] for different models like requirements models.

Lee et al. [154] provide a change impact analysis approach using a goal-driven traceability-based techniques. There is no explicit requirements change classification in the approach although change types such as *modify an existing requirement* and *add a new requirement* are

introduced in the example section of [154]. Instead of providing a requirements change classification, Nurmuliani et al. [192] focus on establishing how practitioners classify requirements change requests. The *Card Sorting*, a knowledge elicitation method, is used to identify categories of change requests in practice. For instance, requirements changes are categorized as *high effort*, *medium effort*, *low effort* and *no effort changes* based on the *magnitude of effort involved* criterion by the practitioners. Harker et al. [107] describe a classification of changing requirements where each changing requirement type could be reformulated as a change type. Lam et al. [150] propose a change maturity model that reflects an organization's capability at managing change. In this maturity model, a change classification is provided with three main types of change: *screen change*, *report change* and *data change*. The change classification in [150] is specialized for Customer Complaints Systems (CCCs). Ackermann and Lindvall [5] classify change requests as *data flow change*, *program flow change* and *application domain change*. Contrary to our approach, none of these change classifications given above except the work in [6] has formal semantics.

5.9.2 Change Impact Analysis in Requirements

A number of approaches in the literature address change impact analysis in requirements. Jonsson and Lindvall [133] present common strategies for change impact analysis from a requirements engineering perspective. They categorize strategies as *automatable* (traceability/dependency analysis and slicing techniques) and *manual* (design documentation and interviews). Automatable impact analysis strategies often employ algorithmic methods for change propagation [133]. Traceability analysis is an automatable strategy that examines relations among all types of software development artifacts. Since our approach analyzes requirements relations for change impact, it can be considered as traceability analysis.

Event-Based Traceability (EBT) [50] supports change impact analysis with automating trace generation and maintenance. In EBT, requirements and other traceable artifacts, such as design models, are linked through publish-subscribe relationship based on the *Observer design pattern* [88]. The main purpose of EBT is to determine the candidate impacted elements and maintain traces for these elements. Contrary to our approach, in EBT all elements directly/indirectly related to the changed element are candidate impacted. EBT does not support change impact alternatives, identification of false positives and consistency checking of changes.

A goal-driven requirements traceability approach is proposed by Lee et al. [154] to analyze requirements change impacts through goals and use cases. Traces among goals and use cases are established and evaluated. Lee et al. provide trace types with definitions but with no formal semantics. Contrary to our approach, this approach does not focus on change

alternatives for propagating a change from one requirement to another. Cleland-Huang et al. [52] introduce another goal-centric approach for managing impact of a change in non-functional requirements. Non-functional requirements and their dependencies are modeled with a Softgoal Interdependency Graph (SIG). The impact detection in [52] is limited to identifying a set of directly impacted SIG elements without any change type.

Ibrahim et al. [121] present an approach for change impact analysis of object oriented software. Change impact analysis is performed from requirements to design, test case or source code. Ibrahim et al., however, do not explain how to propagate a change from one requirement to another requirement. Turver et al. [247] describe a technique dealing with the ripple effects of a change based on a graph-theoretic model. This technique can be applied not only for source code but also for design and requirements documents. The technique, however, calculates the ripple effects by using relations without any semantic information.

O'Neal [195] [196] proposes a change impact analysis method to evaluate requirement changes. Complementary to our approach, O'Neal addresses the identification of the consequences of a change, such as how much change should be done. Hassine et al. [108] provide change impact analysis approach for requirements described as detailed scenarios. Dependencies between scenarios are used to identify the impacted scenarios. However, the approach does not provide any change alternatives for propagation of change. The requirements engineer has to inspect requirements to identify the type of impact without any proposed alternatives.

Cheng et al. [45] propose a method of requirements change management based on keyword mapping. Each requirement is defined as a keyword and a keyword sentence is used to arrange all the keywords according to certain kind of order. When a change request is received for a keyword, the relations of keywords are analyzed for impact analysis. However, the requirements engineer is not supported in how the change is propagated.

Lock et al. [159] [160] [161] provide an approach that integrates different traceability extraction methods (pre-recorded traceability, dependency, plain experience, extrapolation and certainty analysis) to determine impacted requirements. Impact propagation structure, similar to propagation path in our approach, is used with propagation probability to propagate a proposed change from one requirement to another. Contrary to our approach, the only output is the candidate impacted requirements in the impact propagation structure. In addition to candidate impacted requirements, our approach provides change alternatives to be chosen by the requirements engineer.

Lai et al. [148] [149] provide a model-based approach for propagating changes between requirements and design models (particularly activity and sequence diagrams). A change propagation algorithm is proposed to identify and localize the effects of change across requirements and design models. Our approach mainly focuses on change impact analysis in requirements models based on semantics of requirements relations. None of the approaches given above supports consistency checking of requirements changes.

5.9.3 Tool Support

Some requirements management tools support change impact analysis in requirements. The selection of tools is based on INCOSE management tool survey [124].

IBM Rational RequisitePro [119] provides a matrix view to show the requirements relations and their direction between two requirements, or requirements and design elements. When a requirement is changed, relations of the changed requirement are marked as suspect. All requirements directly or indirectly related to the changed requirement are candidate impacted. The requirements engineer has to inspect the candidate impacted requirements to identify changes if there is any. In Borland Caliber [27] change impact analysis is manual. Similar to RequisitePro, Borland Caliber provides traceability matrix and traceability diagram to represent traces where requirements relations are also considered as a trace. Therefore, the requirements engineer should inspect the impacted requirements by using traceability matrix and diagram manually.

TopTeam Analyst [246] supports suspected relations for change impact analysis. However, requirements relations should be manually marked as suspect when a requirement is changed. On the other hand, it is possible to get subscribed to specific elements in artifacts. When one of these elements such as a requirement is changed, the subscribers get a message. The message contains the name of the element, the user who changed the element and a link to the element for a quick inspection.

IBM Telelogic Doors [120] supports a manual analysis of the relations and requirements affected by a change. When a requirement is changed, its relations are marked as suspect automatically. DOORS provides a Change Proposal System (CPS) similar to change impact analysis feature of TRIC. It allows the requirements engineer to investigate, allow or deny change proposals. The requirements engineer can keep an overview of proposed changes and can determine the effect of these changes. However, DOORS does not provide elimination of false positive impacts and any change alternatives for change propagation. None of the industrial tools given above supports consistency checking of requirements changes.

5.10 Conclusions

We presented an approach for change impact analysis in requirements. We provided a classification of requirements changes. The usage of the formal semantics of relations and change types enables new proposed changes to be deduced and contradicting proposed changes to be determined in the requirements model. Most of the approaches and tools like IBM Rational RequisitePro do not focus on formal semantics of requirements relations and change types. With having formal semantics, we provide a more precise change impact analysis in requirements models by supporting change alternatives in change propagation, elimination of false positive impacts and consistency checking of changes. None of the industrial requirements management tools support change impact alternatives and consistency checking of changes. The main advantage of our approach is that propagation alternatives are provided to be chosen by the requirements engineer. By providing change alternatives with impact prediction we determine some of the false positive impacts occurred in most of the industrial tools like IBM RequisitePro.

In this chapter, we answered the part of Research *Question 5* raised in Chapter 1: *How can a change in a requirement be propagated to other requirements and to software architecture? How can we support the requirements engineer and software architect for performing changes? How can we formally check if the evolved architecture satisfies evolved requirements? How can we become sure that traces are up-to-date?* The use of semantics of relations and change types with tool support addresses the propagation of a change from a requirement to other requirements.

There are still open issues. Since we applied the approach to a limited number of requirements in the Course Management System requirements document, the results may not be generalizable. We aim at empirical evaluation of our approach [249] with a quasi-experiment, comparing TRIC with other tools (Microsoft Excel and RequisitePro) for change impact analysis. However, the results of the empirical evaluation are subject to limitations such as low participant representativeness, small sample size, limited comparability of software tools, low participant reliability and training for a new tool.

The definitions of the requirements relations do not give information about the structure of properties in a requirement. For instance, the *contains* relation does not state exactly which property is contained by the containing requirement. The requirements engineer has to inspect the requirements to know this. Therefore, our approach provides change alternatives in change propagation to be chosen by the requirements engineer.

In the current tool support, change propagation alternatives (see Table 5.2) and inconsistencies (see Table 5.3) are implemented in a rule based form. It might be possible to

derive change propagation alternatives and inconsistencies automatically. One possible future work is automatically deriving these propagation alternatives and inconsistencies.

The output of this chapter (requirements change types, proposed changes and propagated proposed changes) is the input for change impact analysis in software architectures. For the evolution of requirements, we will analyze the impact of requirements changes on software architectures. Complete and valid traces between requirements and software architecture are needed in order to propagate changes in requirements to software architectures. In Chapter 6, we provide an approach for generation and validation of traces between requirements and architecture with a tool support. The change impact analysis approach for software architectures is given in Chapter 7. The approach uses the traces with the output of this chapter in order to determine the impacted architectural elements and to fix the software architecture for changed requirements.

Chapter 6

6 Traces between Requirements and Software Architecture

In this chapter, we present an approach for trace establishment based on semantics of traces between Requirements (R) & Architecture (A). Requirements relations and architecture verification techniques are used. We provide a trace metamodel with commonly used trace types. The semantics of traces is formalized in first-order logic. We use the semantics of traces and requirements relations for generating and validating traces with a tool support. The tool provides the following: (1) generation and validation of traces by using requirements relations and/or verification of architecture, (2) generation and validation of requirements relations by using traces. The tool is based on model transformations in ATL and term-rewriting logic in Maude. We illustrate our approach in an example.

6.1 Introduction

In Chapter 5, we presented a change impact analysis approach in requirements models. To overcome the explosion of impacts addressed in [23], we provide semantics of requirements changes together with formal semantics of requirements relations given in Chapter 4. We use the formalization of relations and changes for propagating and consistency checking of proposed changes. Once we analyze the impact of a change in requirements, we need to determine the impact of this change in software architecture by using traces between Requirements (R) and Software Architecture (A). For example, in Figure 6.1, a change in requirement R_2 has a direct impact on architectural component C_2 through the trace between R_2 and C_2 . It may also have an indirect impact on component C_1 through the *refines* relation between R_2 and R_1 , and the trace between R_1 and C_1 . We need complete and valid traces between R&A in order to identify impacted architectural elements for requirements changes.

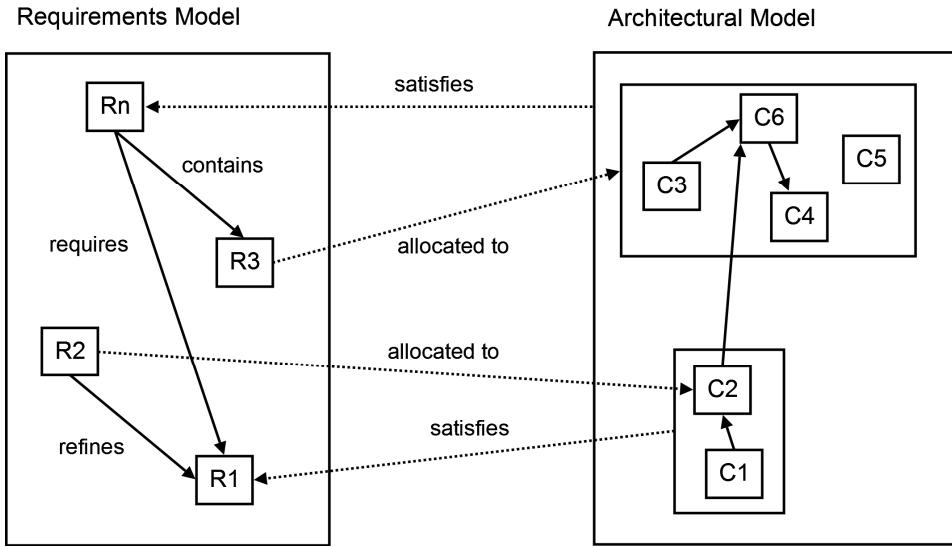


Figure 6.1 Within-Model and Between-Model Traces with Requirements Relation Types and Trace Types between Requirements and Software Architectures

Considerable research has been devoted to relating requirements and design artifacts with source code. Most approaches focus on generating traces between requirements and source code or between design and source code [13] [70] [102] [109]. Less attention has been paid to relating requirements with architecture by using well-defined semantics of traces. Designing architecture based on requirements is a creative and manual process. The software architect can manually assign traces between R&A. Manual trace assignment is time-consuming, expensive and error prone. In most approaches, there is a lack of precise definition of traces between R&A. This lack may cause incomplete and invalid trace establishment for requirements and architecture, thus prohibiting accurate change impact analysis.

In this chapter, we present an approach that provides trace establishment by using semantics of traces between R&A (Requirements and Architecture). Our approach for trace establishment includes trace generation and validation. *Generating traces* is the activity of deducing traces between requirements and architecture based solely on verification of architecture and/or the requirements relations. *Validating traces* is the activity of identifying traces which do not obey trace semantics. We use a trace metamodel with commonly used trace types: *Satisfies* and *AllocatedTo* (see Section 6.3 for details of trace types). The semantics of the traces is provided with a formalization in first-order logic. Software architectures are expressed in Architecture Analysis and Design Language (AADL) [225]. We use dynamic semantics for part of AADL [197] [198] expressed in rewriting logic supported by the Maude language and tools [48] [49]. Semantics of AADL given in Maude enables simulation and

verification of AADL models [221]. For verification of AADL models, we use model checking of the systems' behavior with respect to selected properties [264].

We propose two mechanisms to generate traces between R&A. The first mechanism uses architecture verification techniques. A given requirement is reformulated as a property in terms of the architecture. The architecture is executed and a state space is produced. This execution simulates the behavior of the system on the architecture level. The property derived from the requirement is checked by the Maude model checker. Traces are generated between the requirement and the architectural components used in the verification of the property.

The second mechanism uses the requirements relations together with the semantics of traces. We ensure that the relations between requirements are preserved in their implementation in the architecture. This preservation is also used in the concept of *software reflexion models* where relations between elements in high-level models are preserved in their implementations [185]. Requirements relations are reflected in the connections among the traced architectural elements based on the semantics of traces. Therefore, new traces are inferred from existing traces by using requirements relations. We use semantics of requirements relations and traces to both generate/validate traces and generate/validate requirements relations.

In this chapter, we answer *Research Question 4* (*How to model requirements, software architecture and traces with their semantics for change management?*) and *Research Question 5* (*How can we formally check if the evolved architecture satisfies evolved requirements? How can we become sure that traces are up-to-date?*) raised in Chapter 1. With the approach for trace establishment we address the issues about the use of formal semantics to reason about traces.

Our approach is supported by a tool that uses ATL model transformations [135] [136] in combination with Maude. The tool provides the following: (1) generation and validation of traces by using requirements relations and/or verification of architecture, (2) generation and validation of requirements relations by using traces. We illustrate our approach in an example.

This chapter is structured as follows. Section 6.2 describes the approach. Section 6.3 presents the trace metamodel and definitions of the trace. In Section 6.4, we provide the formalization of the relations. Section 6.5 introduces the example. Section 6.6 describes generating and validating traces based on formal trace semantics. Section 6.7 explains the tool support. Section 6.8 discusses on the approach for the open issues. In Section 6.9, we

illustrate the approach with the Remote Patient Monitoring (RPM) example. Section 6.10 describes the related work, and Section 6.11 concludes the chapter.

6.2 Overview of the Approach

Our approach supports several scenarios with different degrees of automation of trace generation and validation. Figure 6.2 gives the overview of the approach.

Scenario 1: *Generating/Validating traces by using requirements relations.* This scenario takes the requirements model, an initial trace model and constraints in Figure 6.2 as input. The initial traces are assigned by the architect in the input trace model. Traces are generated for requirements which do not have any assigned traces but which are related to requirements with assigned traces. The semantics of trace and requirements relations is used to deduce the new traces. The output trace model contains the generated traces. The requirements relations and the constraints are used to check the validity of the assigned traces in the input trace model. Invalid traces are reported in the output error model.

Scenario 2: *Generating/Validating traces by using verification of architecture.* We check if the requirements are satisfied by the architecture. This is done by reformulating the requirements in terms of logical formulas over the architecture. This scenario takes the reformulated requirement(s), the input trace model and the architectural model as input. To check the formulas we perform architecture simulation and verification in Maude. If the result of the verification is positive, all the architectural elements used in the execution trace⁴ are considered to be related to the requirement with the *Satisfies* traces. Traces are generated accordingly in the output trace model. If a counter example is found, all the architectural elements used in the counter example are considered to be related to the requirement with the *AllocatedTo* traces. The software architect should inspect the input models for errors. The validation phase compares the assigned traces in the input trace model with the architectural elements in the verification output. The invalid assigned traces are reported in the output error model.

Scenario 3: *Generating/Validating traces by using requirements relations and verification of architecture.* This scenario is the combination of the first two scenarios and takes the reformulated requirement(s), the input trace model, the requirements model and the architectural model as input. First, initial traces are generated for the reformulated requirement(s) by using verification of architecture. Then, requirements relations in the input requirements model are used to generate traces for other requirements. The newly generated traces are placed in the

⁴ Execution traces should not be confused with the R&A traces

output trace model. The validation step considers two cases. In the first case, the input trace model is empty. Then, traces generated from the verification output are validated by using the requirements relations. The output is the error model which contains invalid generated traces. In the second case, the input trace model contains assigned traces. New traces are generated from the verification output. The assigned and generated traces are compared for validation with the help of requirements relations. The output error model contains the invalid assigned traces.

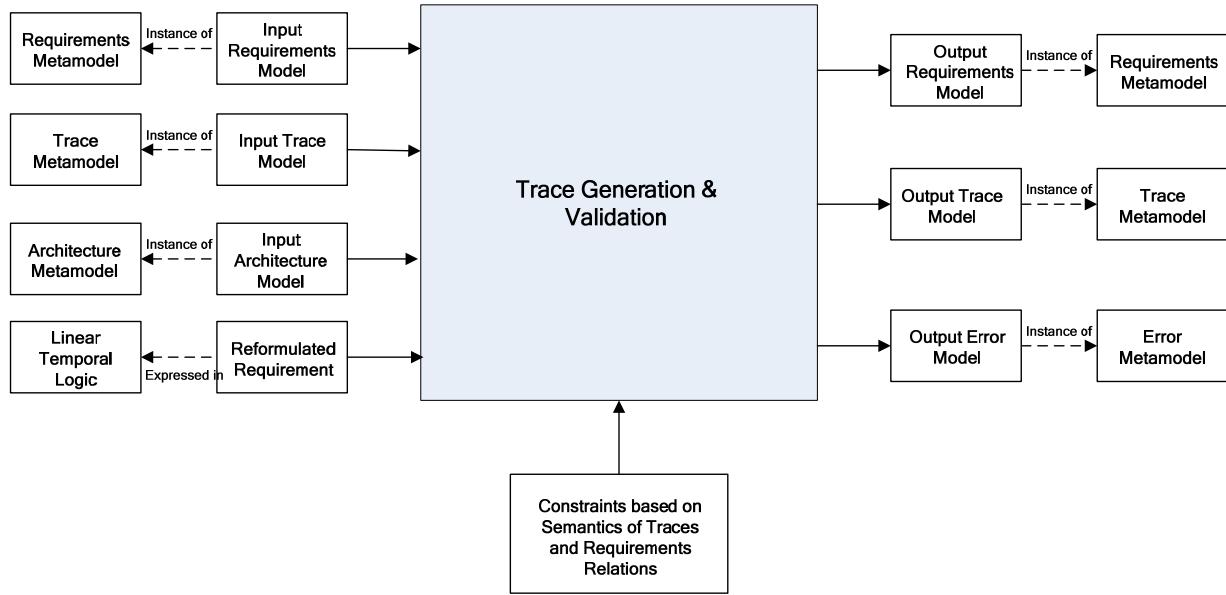


Figure 6.2 Overview of the Approach

Scenario 4: Generating/Validating requirements relations by using traces. The input trace model contains traces which might be either assigned or generated. The relations among architectural elements may reveal new relations, or the lack of relations between the traced requirements according to the constraints based on semantics of traces and requirements relations. For instance, one of the constraints is that if one requirement requires another requirement, there should be, at least, an architectural element that satisfies both requirements. The output requirements model contains the generated requirements relations. The output error model contains the invalid requirements relations in the input requirements model.

We have to note that all generated/invalid traces and requirements relations are suggestions for the architect. They have to be checked by the architect for the final decision. In order to

facilitate the scenarios, we rely on the semantics of requirements and relations previously given in Chapter 4. In addition, in this chapter we successively provide the followings:

- **Trace metamodel.** We use a trace metamodel [66] to structure the traces. The metamodel includes most commonly found entities in literature, and requirements & architecture specific traces (Section 6.3).
- **Semantics of traces.** We formalize traces between R&A by using FOL (Section 6.4).
- **Architecture description and verification facilities.** Software architectures are expressed in Architecture Analysis and Design Language (AADL) [225]. We use formal dynamic semantics for part of AADL [197] [198] given in rewriting logic used in Maude language and tools [48] [49]. The details of the formal semantics of AADL models in Maude can be found in Appendix E. Formal semantics of AADL enables performing simulation and verification of AADL models [221]. For the verification, architectural significant functional requirements are reformulated as formalized scenarios and then properties are checked using linear temporal logic (LTL) [14]. Application of verification techniques for requirements is not the main focus of this chapter. The details can be found in [212].
- **Generating and validating traces.** We use semantics of traces and requirements relations with architecture verification techniques for generating and validating traces (Section 6.6).

We provide tool support and illustrate the feasibility of our approach in an example.

- **Tool support.** We describe the design and implementation of a prototype tool for generating and validating traces based on formal trace semantics (Section 6.7).
- **Running example.** We illustrate the approach with an example (Section 6.9). The example is about requirements and architecture of a Remote Patient Monitoring (RPM) system developed by a company in the Netherlands. An RPM requirements document is used in this chapter as a running example.

6.3 Trace Metamodel

Our trace metamodel defines trace types between requirements and architecture identified in the literature. There are several approaches about transition from requirements to architecture which define trace types. Some of these approaches are summarized in [86] as:

goal-oriented [250], model bridging [103], problem frames [105], use case maps [41], rule-based decision making [158], architecting requirements [157], object-oriented transition [137], and weaving requirements and architecture processes [193]. For example, goal-oriented requirements engineering [250] defines a model for decomposing a system goal into requirements and operationalizations with goal-trees. Operationalizations can be considered as traces between requirements and architecture. Von Knethen et al. [142] classifies traces as *within-model* and *between-model*. Our trace metamodel consists of between-model traces. Figure 6.3 shows our trace metamodel together with parts of requirements and architecture metamodels. The requirements metamodel is the one described in Chapter 4. In the terminology of Von Knethen, requirements relations are *within-model* traces.

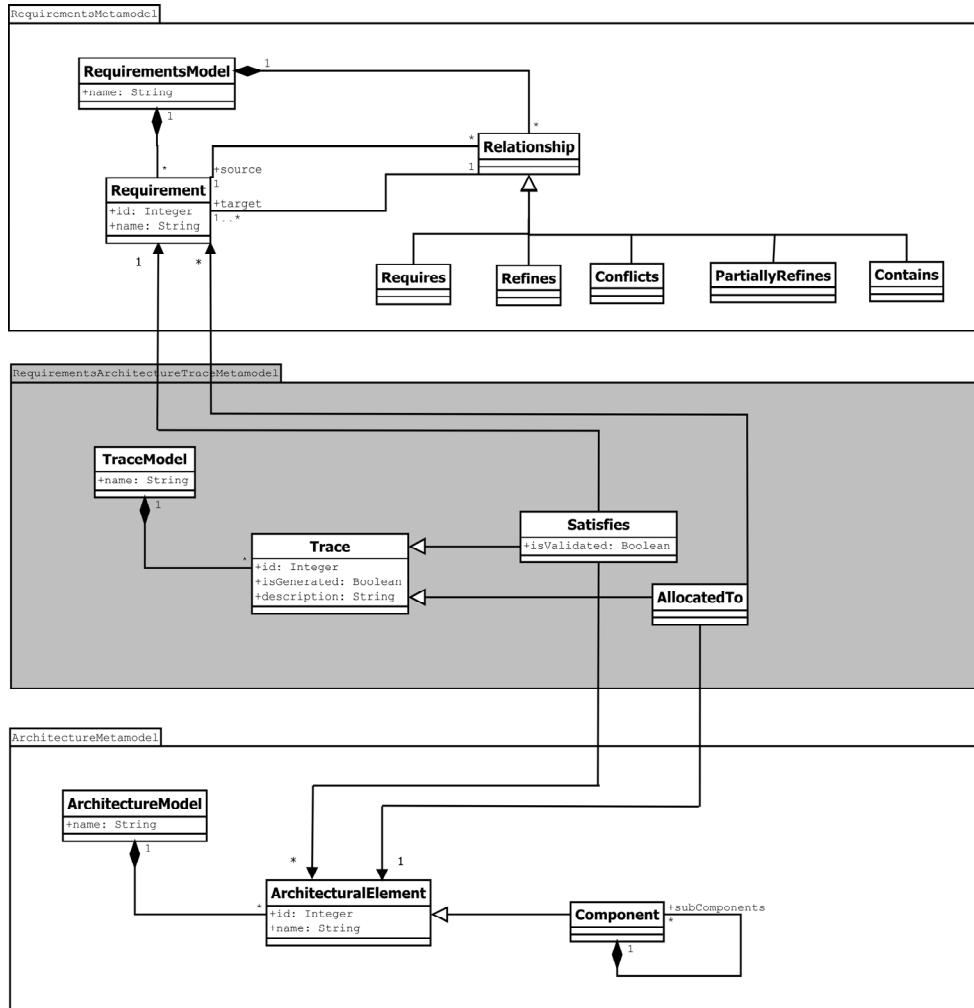


Figure 6.3 Trace Metamodel for Requirements and Architecture

We assume the following definition of software architecture: *A software architecture is a description of the structure of a system, which comprise the software elements, the externally visible properties*

of those elements, and the relationships among them [229]. We use AADL to model the architecture. A fragment of the AADL metamodel is given in Figure 6.3.

We use two types of traces between requirements and architecture: *AllocatedTo* and *Satisfies*. In the literature, these relations are informally defined as follows [200] [215] [258]:

Definition 6.1 *AllocatedTo trace:* A requirement R is *allocated to* a set of architectural elements E if the system properties related to E are supposed to fulfill the system properties given in R.

The architect can track which component will take care of what requirement by using *AllocatedTo* traces [215].

Definition 6.2 *Satisfies trace:* A set of architectural elements E *satisfies* a requirement R if the system properties related to E fulfill the system properties given in R.

A *Satisfies* trace addresses an implication dependency between the system properties given in the requirement and system properties designed in the architecture. The architecture *satisfies* the requirement where the fulfillment of system properties described in the architecture implies the fulfillment of the system properties given in the requirement.

An *AllocatedTo* trace is assigned when the fulfillment of the requirement is expected. A *Satisfies* trace is assigned or generated when the fulfillment of the requirement is present.

The literature proposes several types of traces, which are similar to *Satisfies* and *AllocatedTo* but named differently. For example, Khan et al. [138] propose six types of traces. They differ only in the type of the source requirement. In our approach we abstract from this detail thus keeping the generic types *Satisfies* and *AllocatedTo*. Section 6.10.1 further discusses the trace types found in the literature.

The definitions given above are informal and can be interpreted differently. Since we aim at precise semantics, we formalize trace types in FOL.

6.4 Formalization of Trace Types

In this section we formalize the trace types. In Section 6.4.1 we briefly repeat the definition of requirements as found in Chapter 4. Section 6.4.2 presents the formalization of software architectures. In Section 6.4.3, we introduce the formalization for trace types between requirements and architecture.

6.4.1 Formalization of Requirements

We assume the general notion of requirement being “a property which must be exhibited by a system”. We assume that requirements can always be expressed as a formula in the universal fragment of FOL as $\forall x\varphi$ with φ in conjunctive normal form (CNF). If the formula φ is a closed formula, then the universal quantifiers can be dropped. It is also possible that the formula contains free variables.

6.4.2 Formalization of Architecture

A software architecture model AM is a model conforming to the AADL metamodel. There are different works in the literature [12] [28] [29] [221] that provide a formal semantics of the following notions: *metamodel*, *model*, and *conformance of a model to its metamodel*. We do not repeat the formalization of these notions in this thesis.

We consider the software architecture model AM as an implementation of a property or properties which must be exhibited by a system. The software architecture model AM has architectural elements - the computational units which collectively constitute an architecture. The architectural elements in the subset of AADL that we use are *System*, *Process*, *Thread Group*, *Thread*, *SubProgram*, *Data Store*, *Port*, *Data Access* and *Connector*. For a given property P_A , we are interested in identifying the set of architectural elements E_A that are responsible for fulfilling P_A . We express the property as a formula P_A in any suitable logic such as *Linear Temporal Logic (LTL)* or *Computation-Tree Logic (CTL)*. The property P_A can be checked over the architecture model AM by using architecture verification techniques.

6.4.3 Formalization of Satisfies and AllocatedTo Trace Types

Traces are generally subsets of Cartesian products of sets. We define *Satisfies* and *AllocatedTo* trace types as follows:

$$(74) \quad \text{Satisfies} \subseteq \wp(SAE) \times SR \text{ and } \text{AllocatedTo} \subseteq SR \times \wp(SAE)$$

where SR is the set of requirements in the requirements model RM and SAE is the set of architectural elements in the software architecture model AM . The definition of the *AllocatedTo* trace type formalizes the intuition that a part of software architecture is planned to be an implementation of a set of requirements.

Let R be a requirement and E_A be a set of architectural elements where P_R is a formula in CNF for R and P_A is a formula in LTL. Figure 6.4 gives the schematic view of the relation between P_R and P_A .

We require the following for the *Satisfies* trace type:

E_A Satisfies R iff the following statement holds:

(75) The fulfillment of P_A implies the fulfillment of P_R

This definition of the *Satisfies* trace type formalizes the intuition that a part of software architecture is an implementation of a set of requirements. The set of architectural elements (E_A) fulfills a property (P_A) which is a refinement of a property (P_R) given in a requirement (R). The architectural elements in E_A are in the execution trace of checking P_A . This is explained later. The refinement of P_R to P_A and modeling of the architecture are manual. P_A is considered a refinement because in the general case the software architect makes certain design decisions that narrow the set of systems that satisfy the requirements.

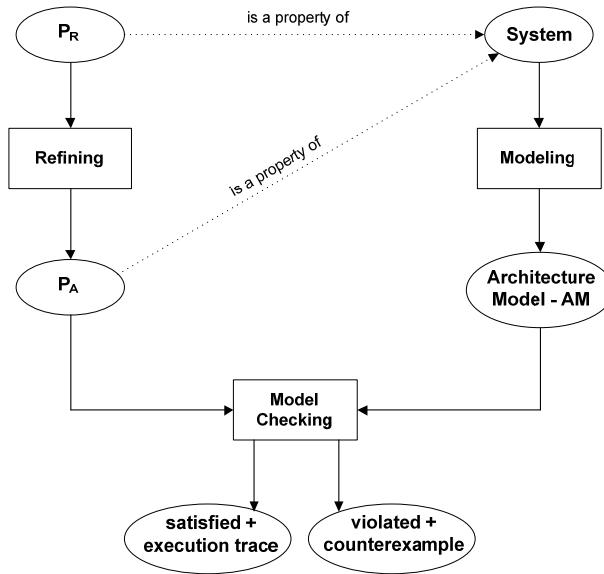


Figure 6.4 Schematic View of the Relation between P_R and P_A

The whole software architecture model implements all the architecturally significant requirements in the requirements model. Architecturally significant requirements play an important role in determining the architecture of the system. Not all requirements have equal significance with regards to the architecture. According to [173], architecturally significant requirements are those that (1) capture essential functionality of the system, (2) exercise many architectural elements, (3) challenge the architecture, (4) highlight identified issues/risks, (5) exemplify stringent demands on the architecture (e.g. performance requirements), (6) are likely to change, and (7) involve communication and synchronization with external systems. Every architecturally significant requirement should be satisfied and

every architectural element should contribute to at least one requirement. We define the *Satisfies* relation between the requirements model RM and the architecture model AM :

The Architecture Model AM satisfies the Requirements Model RM iff the following two statements hold where R is a requirement, SAR is the set of architecturally significant requirements in the requirements model RM , AE is an architectural element and SAE is the set of architectural elements in the architecture model AM :

$$(76) \quad \forall AE(AE \in SAE \rightarrow \exists E_A \exists R(E_A \in \wp(SAE) \wedge AE \in E_A \wedge \text{Satisfies}(E_A, R)))$$

$$(77) \quad \forall R((R \in SAR \wedge \neg \text{refined}(R)) \rightarrow \exists E_A(E_A \in \wp(SAE) \wedge \text{Satisfies}(E_A, R)))$$

$\text{refined}(R)$ is true iff R is refined by one or more requirements. The most refined requirements in the requirements model are the most concrete requirements satisfied by the software architecture.

6.5 Example: Remote Patient Monitoring System

In this section, we introduce the Remote Patient Monitoring (RPM) system as a running example. The example is about requirements and architecture of a RPM system. The RPM system has the following stakeholders: patients, doctors, and the system administrator. The main goal of the RPM system is to monitor the patients' condition such as blood pressure, heart rate and temperature. For instance, the system has to perform a temperature measurement at the patient. The patient carries a sensor for the measurement. Each temperature measurement is transferred to a central system which stores the measurements.

The example system was developed by a company in the Netherlands. The system had already been implemented and running when we started studying the system. The artifacts of the development of the system are the requirements document, source code and test cases. To deploy the example for our approach, we modeled the textual requirements in the RPM requirements document and their relations according to the semantics of requirements relation types.

The requirements model of the RPM system was created in TRIC (see Chapter 4). Some of the requirements in the RPM requirements document can be found in Appendix F. In the following, two requirements are shown: Requirement 6 requires Requirement 3.

Requirement 3 *The system shall measure blood pressure and temperature from a patient.*

Requirement 6 *The system shall store data measured by sensors in the central storage.*

Figure 6.5 shows the part of the requirements model that we created from the RPM requirements document.

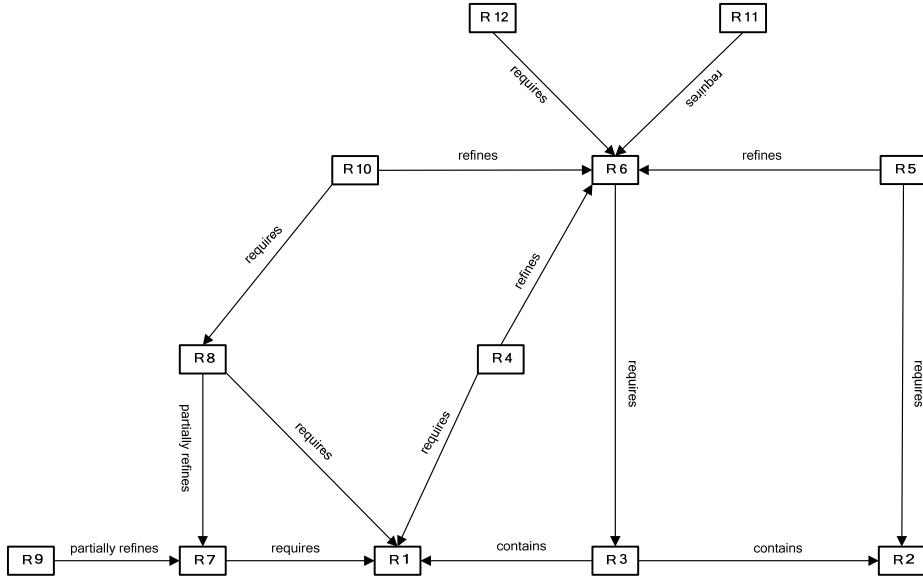


Figure 6.5 Part of Requirements Model for RPM System

The solid arrows indicate the requirements relations given by the requirements engineer. For simplicity, we did not include the inferred requirements relations in Figure 6.5. We constructed the architecture of the system from the source code by reverse engineering. Figure 6.6 gives the overview of the RPM architecture in AADL visual syntax. The graphical notation for architectural elements in AADL is explained in Appendix G. The complete explanations of the abbreviations of the components used in this chapter are given in Appendix H.

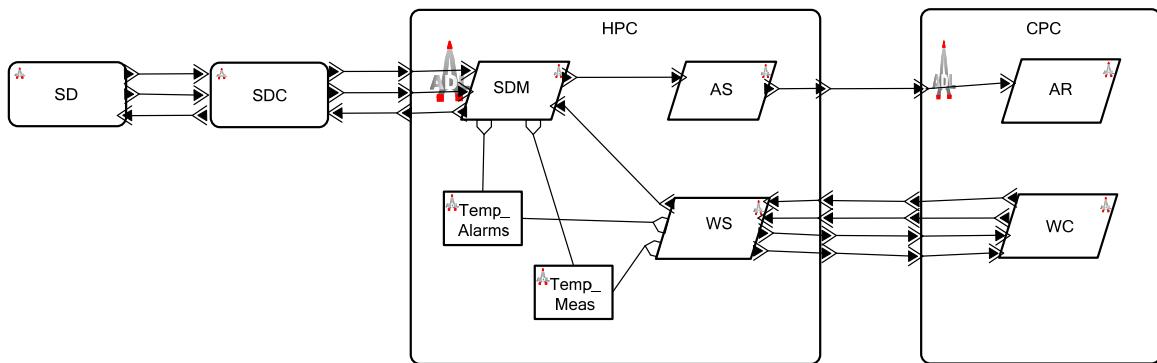


Figure 6.6 Overview of the RPM Architecture

The architecture in Figure 6.6 shows the most abstract components (*system* and *process* in AADL). These components contain other components which we do not represent in Figure 6.6. The *SD* (*Sensor Device*) system component contains the sensors carried by the patient. The sensors perform measurements at a regular interval. If required, the *SD* sends the measurements to the *HPC* (*Host Personal Computer*) system component through the *SDC* (*Sensor Device Coordinator*) system component. The *SDC* is the ZigBee network coordinator. The details of the real coordinating tasks are omitted in the architecture description. The *HPC* consists of the *SDM* (*Sensor Device Manager*), *AS* (*Alarm Service*) and *WS* (*Web Server*) process subcomponents. The *SDM* stores the measurements and generated alarms in the data stores (*Temp_alarms* and *Temp_Meas* for temperature alarms and measurements). The *WS* serves as a web-interface for the doctors. The *AS* forwards the alarms to the *CPC* (*Client Personal Computer*) system component. The *CPC* is used by the doctors to monitor patients. The *AR* (*Alarm Receiver*) process subcomponent in the *CPC* receives the alarms from the *AS* and notifies the doctor about the alarms. The *WC* (*Web Client*) process subcomponent uses the *WS* to retrieve the measurements and alarms stored by the *SDM*.

Figure 6.6 shows only systems and processes in the RPM architecture. AADL provides also support for *thread* and *subprogram* components. The computation of the system is modeled as subprogram and thread behaviour. The current version of the AADL semantics [197] [198] in Maude that we use allows us to model subprogram and thread behaviour by using AADL's behavioral annex with a finite set of states and a set of state variables. The RPM architecture has behavioral annexes for dynamic behaviour of threads in each system component.

The following presents the implementation of the thread in the *SDM* component (*SDM_Thread*) for storing blood pressure measurements. It shows a transition system with state variables where each transition contains a guard (*[sdm_blood_edp2?(inMessage)]* in line 17) on the existence of events/data in the input ports (*sdm_blood_edp2* in line 17), and on the value of the data received (*inMessage* in line 17).

```

1   thread SDM_Thread
2     features
3       sdm_blood_edp2: in event data port Behavior::integer;
4       sdm_blood_strg: out event data port Behavior::integer;
5     properties
6       Dispatch_Protocol => aperiodic;
7   end SDM_Thread;
8

```

```

9   thread implementation SDM_Thread.i
10      annex behavior_specification {**
11          states
12              s0: initial complete state;
13              bloodStored: complete state;
14          state variables
15              inMessage: Behavior::integer;
16          transitions
17              s0 -[sdm_blood_edp2?(inMessage)]-> bloodStored { sdm_blood_strg!(inMessage); };
18          **};
19      end SDM_Thread.i;

```

The thread above has event data ports *SDM_BLOOD_EDP2* in line 3 and *SDM_BLOOD_STRG* in line 4 for blood measurements. Since the *Dispatch_Protocol* of the thread is aperiodic (see line 6), this thread is activated upon receiving input. The thread has states *s0* as the *initial* state in line 12 and *bloodStored* as the *complete* state in line 13. If the thread is in the *s0* state and receives the measurement data at the *SDM_BLOOD_EDP2* event data port, then the received data is stored in the *SDM_BLOOD_STRG* data port and the *bloodStored* state is reached (see line 17).

6.6 Generating and Validating Traces

An important element of our approach is the ability to verify architectures thanks to the semantics definition of AADL in Maude. Both the generation and validation activities depend on it. This section describes how the results from the verification together with semantics of traces and requirements relations are used. Section 6.6.1 explains the verification of architecture for functional requirements in Maude. Section 6.6.2 gives the details of the trace generation by using requirements relations and verification results. In Section 6.6.3, we illustrate trace validation.

6.6.1 Verification of Architecture for Functional Requirements

We limit ourselves to verification of functional requirements only. The purpose of the verification is to check if requirements are correctly implemented in the architecture. We use model checking for verification of AADL models (see [264] for model checking). Verification results are used in both trace generation and trace validation as an input (Scenario 2 and Scenario 3 in Section 6.2). Figure 6.7 illustrates the verification of architecture for functional requirements.

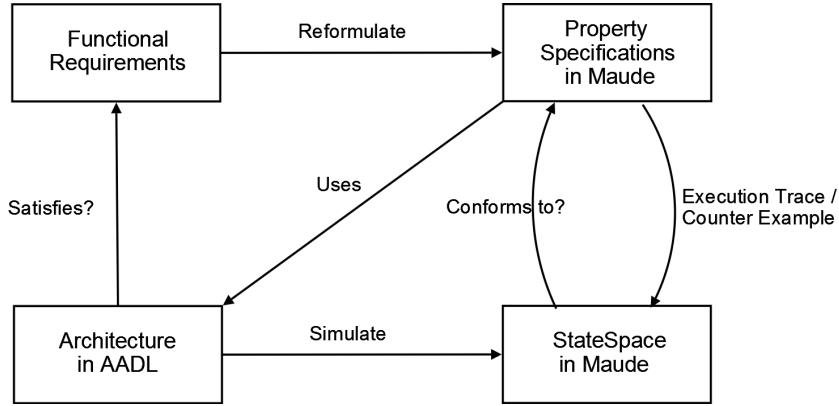


Figure 6.7 Verification of Architecture for Functional Requirements

The output of the verification is represented by the *Satisfies* and *ConformsTo* relations in Figure 6.7. *ConformsTo* implies that the state space captures the specified properties. We have the following artifacts in the process of verification of architecture:

- *Functional Requirements*. Requirements which describe the functions that the system is to execute; for example, formatting some text or receiving data.
- *Architecture in AADL*. The architecture of the system to be built. It plays the role of the solution for the problem defined by the requirements.
- *Property Specifications in Maude*. The formal description of the required behavior of the architecture. The requirements are reformulated as properties in terms of the solution, which is the architecture (*reformulate* and *uses* in Figure 6.7). These properties are checked for the architecture by the model checker. The requirement is first described as a formalized scenario, and then described as property specification [32] [209] [212]. The formalized scenario is a pair of predicates $\langle \text{pre}, \text{post} \rangle$ encoding the precondition *pre* and the postcondition *post* for the architecture. The property specification uses any logic such as *Linear Temporal Logic (LTL)*, *First-Order Logic (FOL)*, or *Computation-Tree Logic (CTL)*. In the tool, we use the formal analysis features of Maude. Maude provides model checking with LTL which is a logical formalism that is suited for specifying Linear-Time properties (see [14] for the details of Linear-Time properties and LTL). In our approach, linear-time properties are formalized first as a scenario and then as an LTL formula.
- *State Space in Maude*. The presence of a dynamic semantics specification of AADL in Maude makes the architectural models executable. The architecture is executed and a state space is produced (*simulate* in Figure 6.7). This execution simulates the behavior

of the system on the architecture level so that it can be studied to see how the system will work. Discrete event simulation, which introduces the notion of events, states, and state space, is used. A state describes the loci of data values within the architecture. Two states are connected by a transition and all states are captured by the state space. The result of the verification, which might be a counter example or an execution trace, is used to generate and validate traces. An execution trace is the ordered set of states which are generated where the reformulated requirement is satisfied. Counter example is the ordered set of states which are generated where the reformulated requirement is not satisfied.

We use the formal semantics of behavioral AADL models in Maude implemented by Olveczky et al. [197] [198]. Since the focus of this chapter is not verification and simulation, we do not give details of the AADL semantics in this chapter. This is itself a non-trivial topic and subject of another work. The AADL semantics in [197] [198] can be found in Appendix E.

Example: Reformulation of Requirements

We explain the reformulation of requirements as property specifications in Maude with the following requirement from the RPM requirements document explained in Section 6.9 and given in Appendix F.

Requirement 5 *The system shall store patient blood pressure measured by the sensor in the central storage.*

The reformulation of Requirement 5 has two steps. Requirement 5 is first reformulated (*reformulate* in Figure 6.7) as a formalized scenario in terms of solution domain – the RPM architecture (*uses* in Figure 6.7). The formalized scenario is a pair of predicates $\langle \text{pre}, \text{post} \rangle$ encoding the precondition *pre* and the postcondition *post* for a dataflow in the architecture (see Figure 6.6 in Section 6.5). Figure 6.8 is the part of the RPM architecture developed for the system property given in Requirement 5.

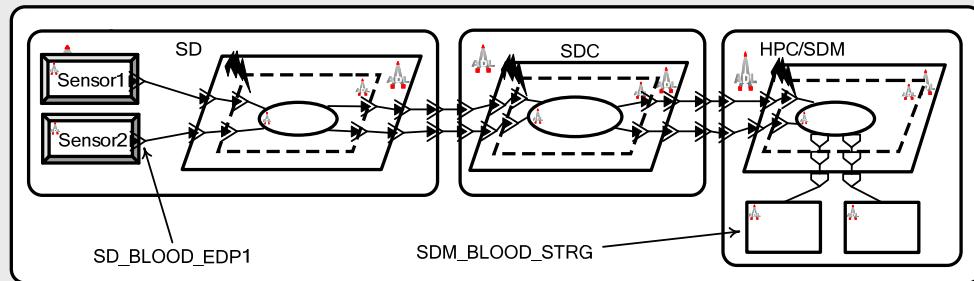


Figure 6.8 Part of the RPM Architecture

Requirement 5 is reformulated as a formalized scenario in terms of solution domain.

Formalized Scenario: (*contains(SD_BLOOD_EDP1, DI)*), (*contains(SDM_BLOOD_STRG, DI)*)

According to the formalized scenario, if the data instance *DI* is contained by the data port *SD_BLOOD_EDP1* of Sensor 2 (*SD* component in Figure 6.8), then the data instance *DI* is stored in the data store *SDM_BLOOD_STRG* of the component *SDM* after executing the architecture (see Figure 6.8).

The dynamic behavior of a thread is defined in AADL using AADL's behavioral annex with a finite set of states and a set of state variables. In the RPM architecture, the subprogram execution for storing the blood pressure data in the central storage is implemented as a state transition system in the thread *sdmTh* (see Section 6.5). The *sdmTh* thread has states *bloodStored*, *temperatureStored*, *highTemperature*, *lowTemperature* and *idle*. When the data instance *DI* is stored in the data store *SDM_BLOOD_STRG* of the component *SDM*, the state of the *sdmTh* thread in the state transition system is set to the *bloodStored* state.

The formalized scenario is the first step to reformulate the requirement in terms of solution domain. The next step is to construct the appropriate logic expression for the formal analysis in Maude. The following is the LTL formula derived from the formalized scenario:

LTL formula in Maude: (*mc initializeThreads({ MAIN system Wholesys . imp }) |=u <> ((MAIN -> hpc -> sdm -> sdmTh) @ bloodStored)*.)

The formula states that if the data instance *DI* is contained by the data port *SD_BLOOD_EDP1* of Sensor 2, then eventually in the future the state in the state transition system in the *sdmTh* thread is set to the *bloodStored* state (the data instance *DI* is stored by the data store *SDM_BLOOD_STRG* of the *SDM* component). Please note that the data instance *DI* is created in the initial state by a test thread in the RPM model. Therefore, the LTL formula does not explicitly indicate the data instance *DI* and the data port *SD_BLOOD_EDP1* of Sensor 2. The formula creates the initial state instead.

The *initializeThreads({ MAIN system Wholesys . imp })* creates the initial state where the data instance *DI* is contained by the data port *SD_BLOOD_EDP1* of Sensor 2. The *MAIN -> hpc -> sdm -> sdmTh* denotes the full component name of the *sdmTh* thread component. The *@bloodStored* states that the state of the *sdmTh* thread is the *bloodStored*. The '*<>*' in the LTL formula states '*eventually in the future*'. The LTL formula can be checked on the generated state space in Maude.

The LTL formula derived from the formalized scenario ($\text{contains}(SD_BLOOD_EDP1, DI)$, $(\text{contains}(SDM_BLOOD_STRG, DI))$ is the property P_A in our formalization of architecture. From the formalization we know that if P_A holds, then P_R given in the requirement also holds.

6.6.2 Generating Traces

Generating traces aims at deducing traces between requirements and architecture based solely on verification of architecture and/or the requirements relations in the requirements model. The approach does not need initial traces to generate new traces (see Scenario 2 in Section 6.2).

The approach uses the result of the verification of architecture. If the verification is successful, the architecture satisfies the requirement. According to the semantics of trace types, the *Satisfies* trace is generated between the architectural elements in the execution trace and the requirement. These elements collectively satisfy the requirement and form the part of the architecture to which the requirement is traced. A counter example means that although the requirement is *allocated to* the architectural elements, the architecture does not satisfy it. The *AllocatedTo* trace can be generated but the *Satisfies* does not hold. We modified the transition rules in Maude to be able to record the architectural elements matched by the transition rules. These matched elements are the used architectural elements during the verification of architecture. These elements correspond to E_A in the formalization of architecture. We modified the AADL metamodel and included an attribute called *Used* to the component classes in the AADL metamodel. Each transition rule sets the attribute *Used* of the architectural element matched in the transition rule to *True*. The details of the implementation of the approach are given in Section 6.7.

The output of the verification for an LTL formula is true or false with a counter example. If the verification returns false with a counter example, the field *used* of the architectural elements matched by the transition rules is set to *true* in the last state of the counter example. To get the execution trace where the requirement is satisfied, we use the *search* command in Maude which allows exploring the reachable state space. The *search* command returns the execution trace where the requirement is satisfied. There might be multiple execution traces where the requirement is satisfied. In this case, the *Satisfies* trace is generated between the architectural elements in each execution trace and the requirement.

A requirement may describe multiple system properties and/or a complex system property amenable to decomposition. In our approach it is not possible to explicitly state which property in the complex requirement fails. The requirements engineer should decompose the

requirement into sub-parts (by using the *Contains* relation) until each requirement describes only one property which can be given as a single LTL formula.

The second way to generate traces is to use the requirements relations (see Scenario 1 and Scenario 3 in Section 6.2). The constraints about traces in Figure 6.9 are derived from the intuition about the semantics of trace types and semantics of requirements relations. The constraints ensure that requirements relations are preserved in the architecture by the satisfying elements. The constraints are also used to generate requirements relations from traces (see Scenario 4 in Section 6.2).

Please note that the constraints are given for the *Satisfies* traces in Figure 6.9. The same constraints are valid also for the assigned *AllocatedTo* traces. The constraint in Figure 6.9(a) states that the intersection of sets of architectural elements that satisfy two requirements where one requires another one is non-empty. In Figure 6.9(b), it is stated that architectural elements that satisfy the refining requirements also satisfy the refined requirement. Constraints similar to the one in Figure 6.9(b) are valid for traces with the *Contains* and *Partially Refines* relations (see Figure 6.9(c) and Figure 6.9(d)).

In order to generate the *Satisfies* traces for R_1 in Figure 6.9(b)(c) and (d), all other requirements (R_2, R_3, \dots, R_k) should be satisfied by the architecture. For instance, if one of the refining requirements (R_2, R_3, \dots, R_k) in Figure 6.9(d) is not satisfied by the architecture, the refined requirement (R_1) is also not satisfied by the architecture. Therefore, there is no *Satisfies* trace for the refined requirement. The partial refinement might not be complete. In this case, even if all refining requirements are satisfied, the *Satisfies* trace is generated only if it is confirmed that the unrefined properties are also satisfied. The *Satisfies* trace is generated for the unrefined properties in R_1 by using the verification of architecture.

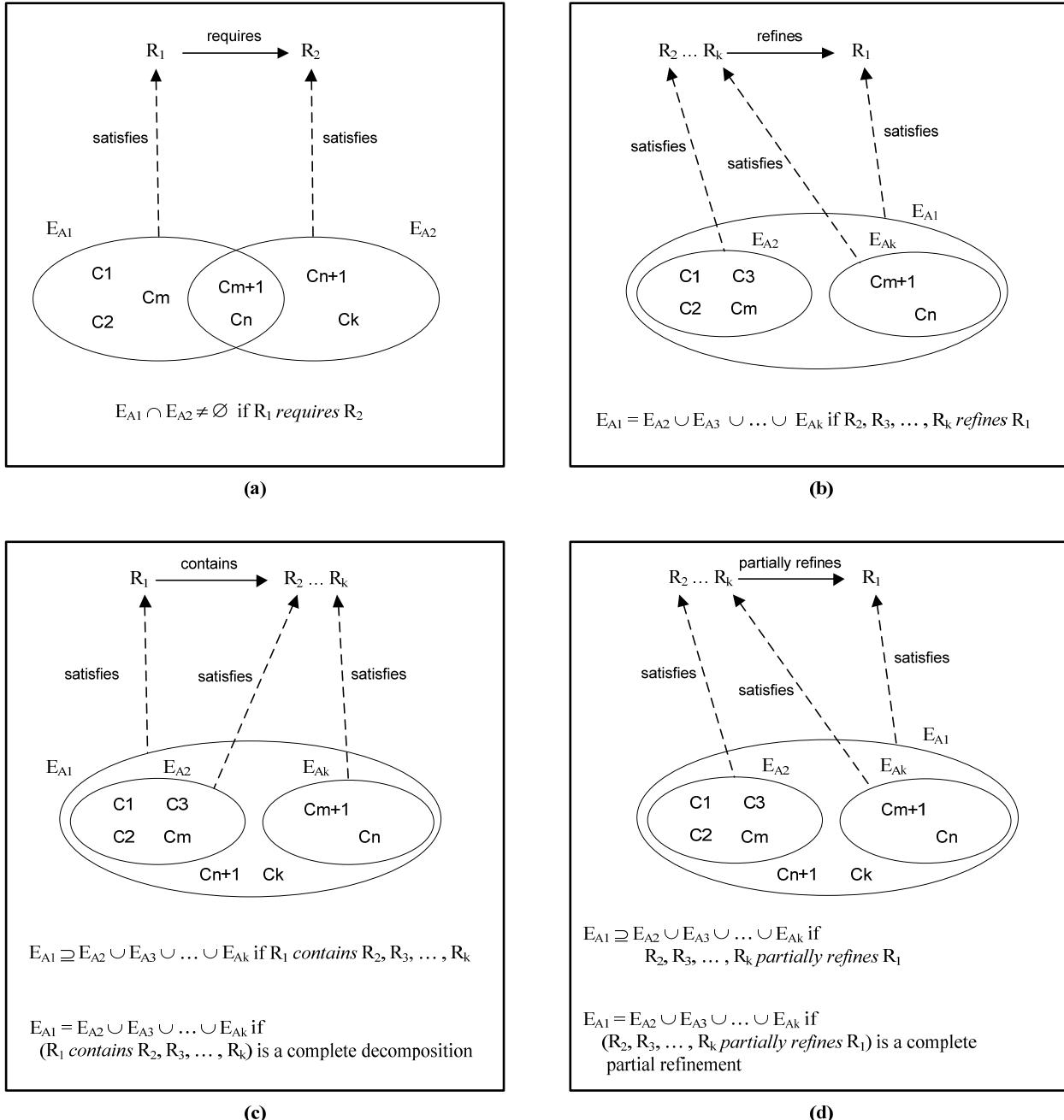


Figure 6.9 Constraints based on Semantics of Traces and Requirements Relations

The following is an example for generation of traces by using verification of architecture.

Example: Generation of Traces by Using Verification of Architecture

In Section 6.6.1, we give an example about the reformulation of requirements as property specifications in Maude for Requirement 5. In this section, we explain how to generate traces

by using the verification of architecture for Requirement 5 (see Scenario 2). The output of the reformulation of Requirement 5 is an LTL formula given below.

LTL formula in Maude: $(mc \text{ initializeThreads}(\{ \text{MAIN system Wholesys . imp } \}) |=u <> (\text{MAIN} \rightarrow hpc \rightarrow sdm \rightarrow sdmTh) @ \text{bloodStored}.)$

After executing Maude, the formula is true which means that Requirement 5 is satisfied by the architecture. As we explained before, the LTL formula does not return the execution trace where the requirement is satisfied. Therefore, we run the *search* command in Maude in order to get the execution trace for Requirement 5:

Search Command in Maude:

(utsearch [1]

initializeThreads(\{ MAIN system Wholesys . imp \}) => {C:Configuration}*

such that

((location of component (MAIN -> hpc -> sdm -> sdmTh) in C:Configuration) == bloodStored .)

In the *search* command above, the *initializeThreads(\{ MAIN system Wholesys . imp \})* creates the initial state where the data instance *DI* is contained by the data port *SD_BLOOD_EDP1* of Sensor 2. The *((location of component (MAIN -> hpc -> sdm -> sdmTh) in C:Configuration)* returns the state in the transition system in the *sdmTh* thread, which should be the *bloodStored* state ('== *bloodStored*). ' $=>*$ ' in the command indicates the form of the rewriting proof from the initial state until the state where the state in the transition system in the *sdmTh* thread is the *bloodStored* state. Then, the *search* command tries to reach that state from the initial state.

The *search* command returns the execution trace where the reformulated requirement is satisfied. The field *used* of the architectural elements matched by the transition rules is set to *true* in the last state of the counter example. Therefore, our tool (explained in Section 6.7) generates the *Satisfies* trace between Requirement 5 and the architectural elements used in the execution trace. Figure 6.10 shows the generated *Satisfies* trace for Requirement 5.

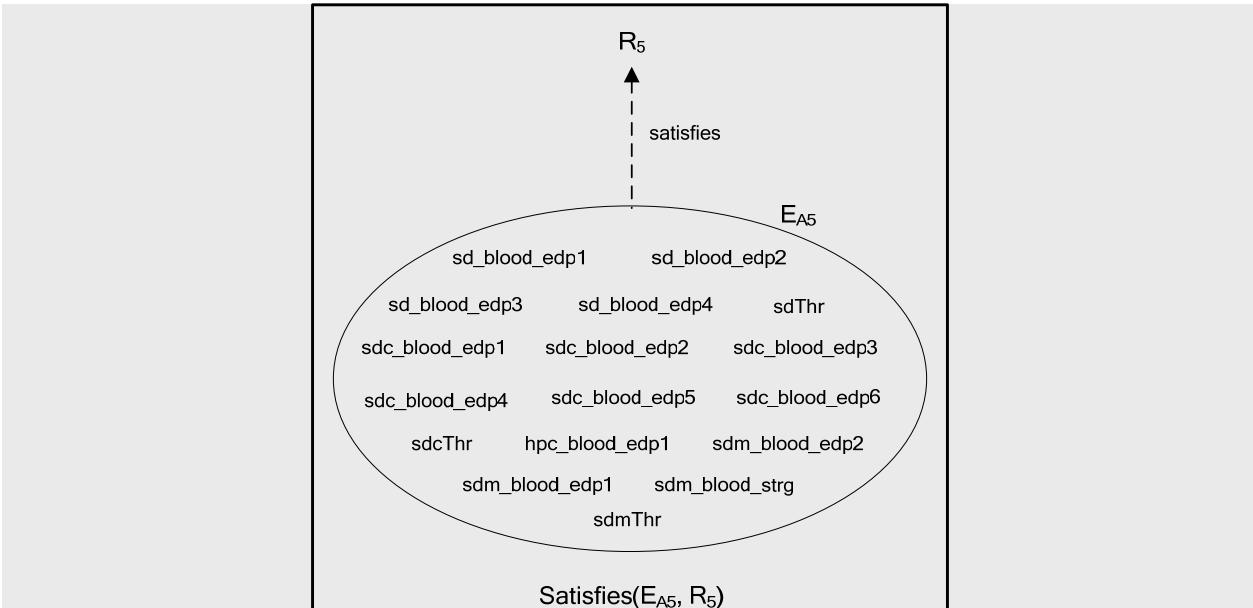


Figure 6.10 Generated ‘Satisfies’ Trace for Requirement 5 by Using Verification Results

In this example, we only explained trace generation by using verification results. Other trace generation scenarios in Section 6.2 are illustrated with other examples in Section 6.9.

The verification result, and therefore the traces, depends on the reformulation of the requirement to be checked. The architect needs to consider potential false positive and missed traces. Such traces are defined in relation to the set of actual traces, which is the *golden standard* for a pair of requirements and architecture. Figure 6.11 gives the classification of traces based on the relation between the actual and the generated traces for a requirement.

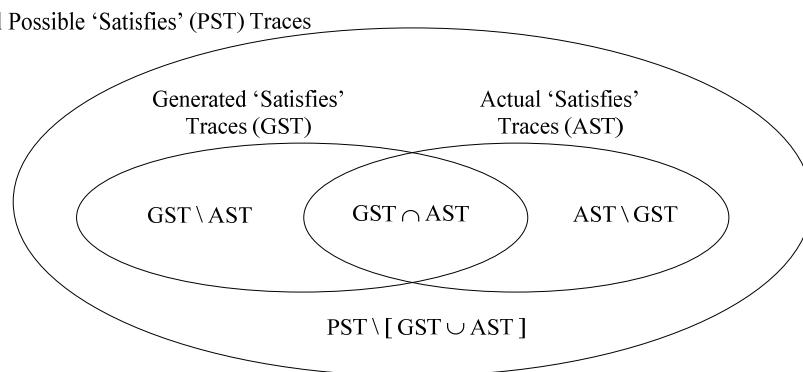


Figure 6.11 Venn Diagram for Generated and Actual Satisfies Traces for a Requirement

The interpretation of Figure 6.11 is given in a confusion matrix [78] in Table 6.1.

Table 6.1 Confusion Matrix of Generated and Actual Traces for Satisfies Relation

		Actual ‘Satisfies’ Traces (AST)	
Generated ‘Satisfies’ Traces (GST)		True Positive	False Positive
Generated ‘Satisfies’ Traces (GST)	True Negative	True Negative	
	False Negative		True Negative

- $(GST \cap AST)$ is True Positive. Generated traces which are also actual.
- $(GST \setminus AST)$ is False Positive. Generated traces which are not actual.
- $(AST \setminus GST)$ is False Negative. Actual traces which are not generated.
- $(PST \setminus (GST \cup AST))$ is True Negative. Traces which are not actual and not generated.

Misinterpretation of the requirement and wrong reformulation are the causes for false positive and false negative traces. In case of ideal models and correct reformulation the generated traces are the actual traces.

6.6.3 Validating Traces

Validation aims at identifying the traces which do not obey the trace semantics. This helps the elimination of false positive traces in Table 6.1. The approach identifies traces or requirements relations which violate the constraints in Figure 6.9. Validation by using requirements relations can be used in two ways (see Scenario 1 and Scenario 4). First, the architect may conclude that an invalid trace is a true positive and then he reconsiders the requirements relations (Scenario 4). Second, the architect concludes that requirements relations are all valid, then, he/she identifies the invalid traces (Scenario 1).

Our approach also provides validation of traces by using verification results (see Scenario 2 and Scenario 3). The architect assigns some *AllocatedTo* traces while creating the architecture (Scenario 2). In order to ensure that the architecture satisfies the requirements, the verification of architecture is performed. For the requirements satisfied by the architecture, the *Satisfies* traces are generated. The assigned *AllocatedTo* traces and the generated *Satisfies* traces for a requirement are validated based on the comparison of traces in Figure 6.12.

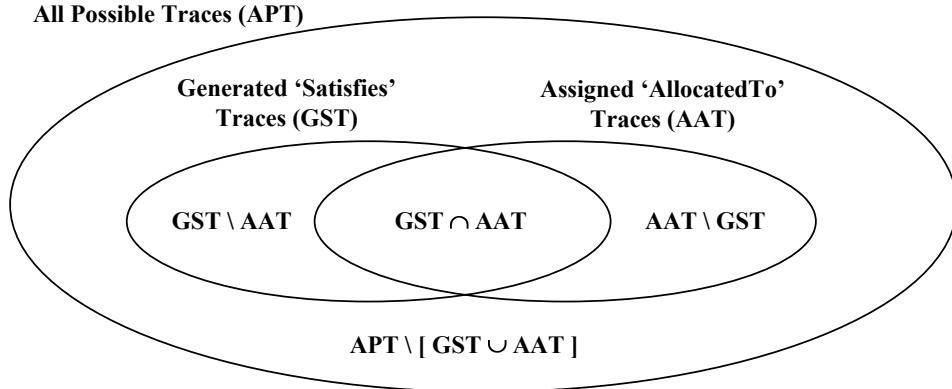


Figure 6.12 Venn Diagram for Generated ‘Satisfies’ and Assigned ‘AllocatedTo’ Traces for a Requirement

The software architect should check the difference of the sets ($GST \setminus AAT$ and $AAT \setminus GST$) and conclude about the validity of traces.

- If $(GST \setminus AAT)$ is non-empty, then either some of the generated *Satisfies* traces ($GST \setminus AAT$) are false positives or some of the traces are missed while assigning the *AllocatedTo* traces. If the software architect concludes that some of the generated *Satisfies* traces ($GST \setminus AAT$) are false positives, then misinterpretation of the requirement and/or wrong reformulation might be the causes of invalid trace generation.
- If $(AAT \setminus GST)$ is non-empty, then either some of the assigned *AllocatedTo* traces ($AAT \setminus GST$) are false positives or some of the *Satisfies* traces are missed while generating the *Satisfies* traces. If the software architect concludes that some of the traces ($AAT \setminus GST$) are missed while generating the *Satisfies* traces, misinterpretation of the requirement and wrong reformulation are the causes of the missing *Satisfies* traces.

For the requirements which are not satisfied by the architecture, the *AllocatedTo* traces are generated from the counter example. The assigned and generated *AllocatedTo* traces for a requirement are validated based on the comparison of traces in Figure 6.13 which is similar to the comparison table in Figure 6.12.

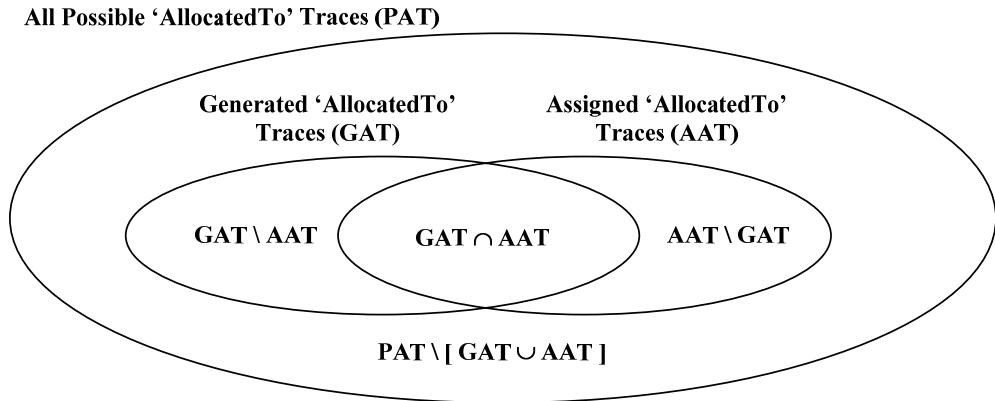


Figure 6.13 Venn Diagram for Generated and Assigned ‘AllocatedTo’ Traces for a Requirement

The software architect should check the difference of the sets ($GAT \setminus AAT$ and $AAT \setminus GAT$) and conclude about the validity of traces.

- If $(GAT \setminus AAT)$ is non-empty, then some of the generated *AllocatedTo* traces ($GAT \setminus AAT$) are false positives or some of the traces are missed while assigning the *AllocatedTo* traces. If the software architect concludes that some of the generated *AllocatedTo* traces ($GAT \setminus AAT$) are false positives, then misinterpretation of the requirement and/or wrong reformulation might be the causes of having a counter example and invalid trace generation.
- If $(AAT \setminus GAT)$ is non-empty, then either some of the assigned *AllocatedTo* traces ($AAT \setminus GAT$) are false positive or some of the traces are missed in the trace generation. If some of the traces are missed in the trace generation, then misinterpretation of the requirement and/or wrong reformulation might be the causes of having a counter example and missing traces.

The following is an example of validation of traces by using verification of architecture.

Example: Validation of Traces by Using Verification of Architecture

In Sections 6.6.1 and 6.6.2, we give examples about the reformulation of requirements and generation of traces for Requirement 5. In this section, we explain how to validate traces by using the verification of architecture for Requirement 5 (see Scenario 2).

The example in Section 6.6.2 shows the generated *Satisfies* traces for Requirement 5. Figure 6.14 gives the generated *Satisfies* and assigned *AllocatedTo* traces for Requirement 5.

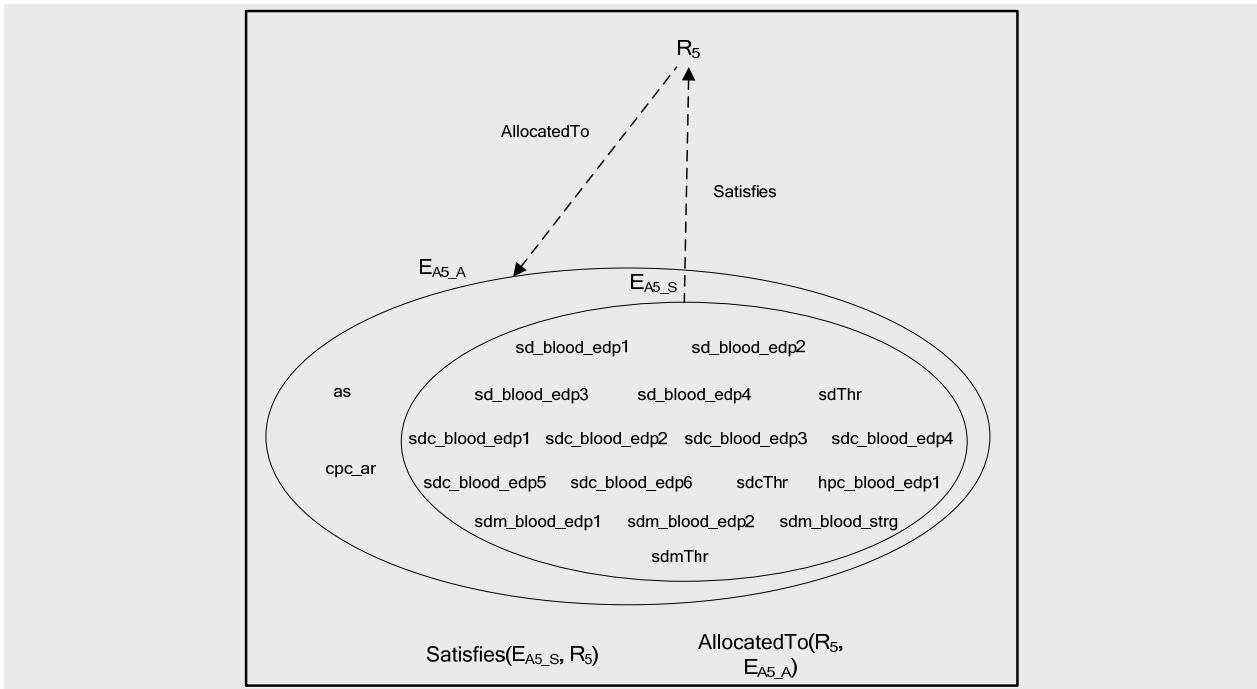


Figure 6.14 Generated ‘Satisfies’ and Assigned ‘AllocatedTo’ Traces for Requirement 5

The traces in Figure 6.14 are validated according to the Venn diagram in Figure 6.12 (see Scenario 2). Although Requirement 5 is allocated to the components *AS* and *CPC_AR*, these components are not involved in the *Satisfies* traces. We concluded that the two *AllocatedTo* traces to the components *AS* and *CPC_AR* are false positives.

In this example, we only explained trace validation by using verification results. Other trace validation scenarios in Section 6.2 are illustrated in Section 6.9.

6.7 Tool Support

We built a tool for generating and validating traces between R&A based on formal trace semantics. In this section, we give the details of the tool. In Section 6.7.1, we depict the usage of the tool in the context of a modeling process. Section 6.7.2 gives the architecture of the tool. Section 6.7.3 describes the main features of the tool with some screenshots. Section 6.7.4 evaluates the tool.

6.7.1 The Modeling Process

The tool is used in the context of a modeling process for trace generation and validation. Figure 6.15 gives a UML activity diagram of the process.

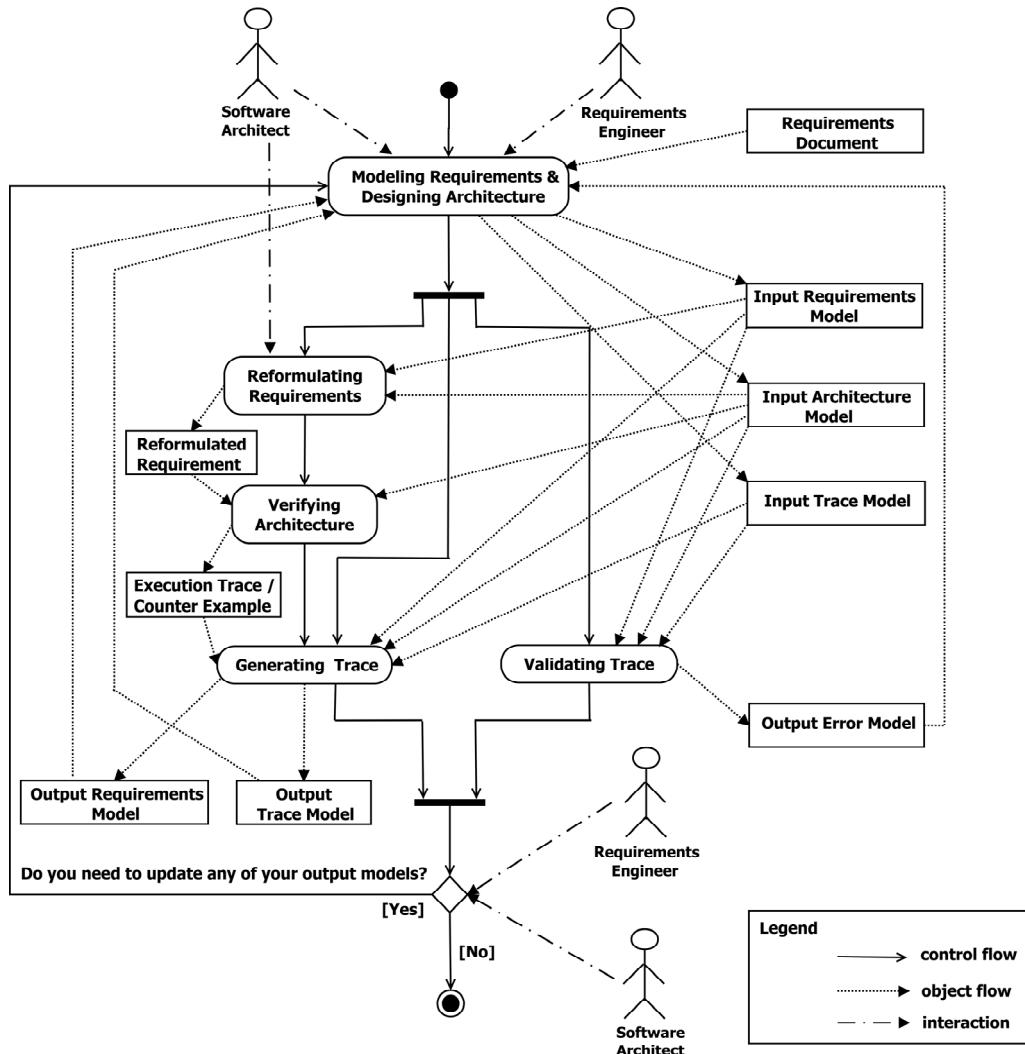


Figure 6.15 Modeling Process with Trace Generation and Validation

The process in Figure 6.15 consists of the following activities:

Modeling Requirements & Designing Architecture: This activity takes the requirements document and produces the input requirements model, input architecture model and input trace model. The software architect assigns some initial traces between requirements and architecture.

The modeling process is separated into three activities: *reformulating requirements*, *generating trace* and *validating trace*.

Reformulating Requirements: This activity takes the input requirements model and input architectural model and produces the reformulated requirement as output. The software architect reformulates the requirements in terms of logical formulas over the architecture.

Verifying Architecture: This activity takes the input architectural model and the reformulated requirement, and produces an execution trace or a counter example (see Section 6.6.1). The activity checks whether the requirements are satisfied by the architecture. It is done automatically in Maude.

Generating Trace: This activity takes the input trace, requirements and architecture models with the output of verifying the architecture and produces the output trace model and requirements model. The activity is automatic. If the activity uses only requirements relations in the requirements model and initial traces in the input trace model, the activity is performed after the activity *modeling requirements & designing architecture*.

Validating Trace: This activity takes the input trace model, input requirements model, input architecture model and produces an output error model. The activity is automatic. However, the interpretation of the errors in the trace model should be done manually by the software architect.

Iterating: The process given in Figure 6.15 is iterative. The requirements engineer and/or the software architect may return to the modeling requirements & designing architecture activity in order to fix requirements relations and/or traces in the output models. If there is no need to update the models, the process is terminated.

6.7.2 Tool Architecture

The tool contains five components (rounded boxes in Figure 6.16): (a) the *Model Checker in Maude*, (b) the *Trace Generator using Verification Results in ATL*, (c) the *Trace Generator using Requirements Relations in ATL*, (d) the *Trace Validator in ATL*, and (d) the *Requirements Relation Generator using Traces in ATL*.

Model Checker in Maude: The input for architecture verification component is the input architecture model and the requirement(s) reformulated in LTL. This component is used in the trace generation part of Scenario 2 and Scenario 3 (see Section 6.2). The verification and simulation are performed by the model checker and the rule execution engine of Maude. The architectural model originally expressed in AADL is transformed to a Maude term [182]. The AADL metamodel is encoded as a set of sorts. The dynamic semantics of AADL is given in rewriting rules [197] [198]. Requirements are reformulated as LTL formulas, the language supported by Maude checker.

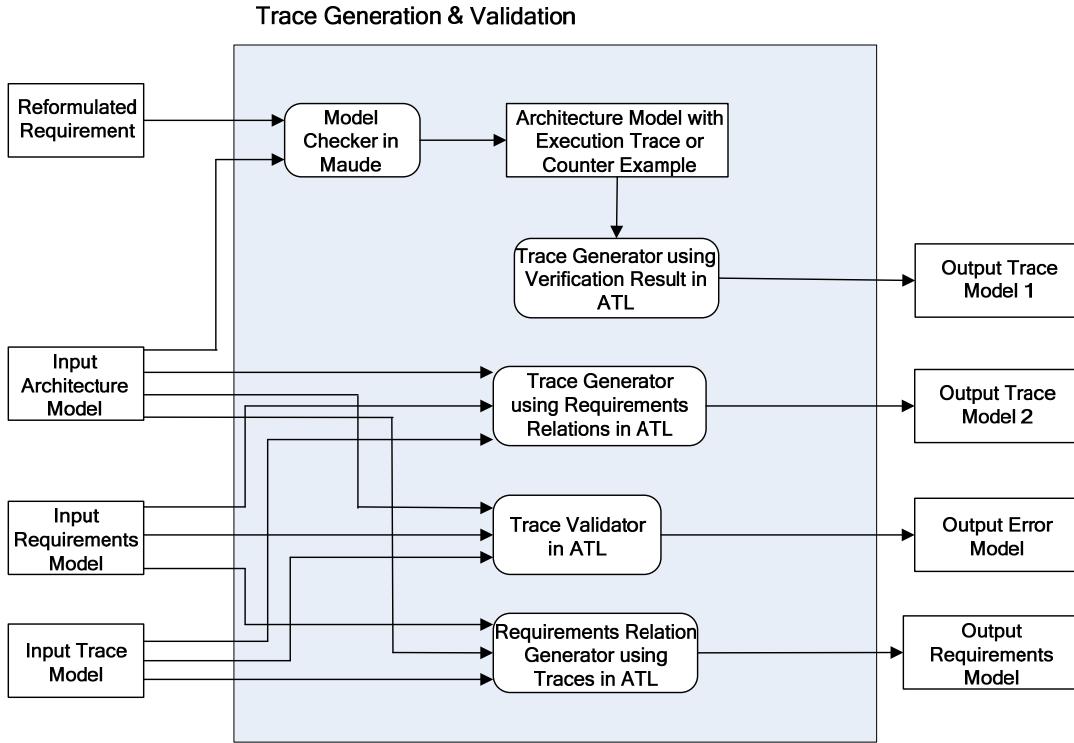


Figure 6.16 Overview of the Tool

Trace Generator using Verification Result in ATL: The input of the component is the execution trace and counter example. The component is implemented as an ATL transformation. If the verification result is an execution trace, the *Satisfies* traces are generated between the checked requirement(s) and the architectural elements in the execution trace. If the verification result is a counter example, the *AllocatedTo* traces are generated between the checked requirement(s) and the architectural elements marked in the counter example. The result is the *Output Trace Model 1*.

Trace Generator using Requirements Relations in ATL: The input of the component is the *Input Architecture Model*, the *Input Trace Model*, and the *Input Requirements Model*. The component is used in the trace generation part of Scenario 1 and Scenario 3. It is implemented as an ATL transformation. The component generates new traces based on the requirements relations in the *Input Requirements Model* and the constraints in Figure 6.9. The output is the *Output Trace Model 2*.

For output of the two trace generator components, we use two different output trace models in order to state that the outputs do not have to be the same. In the generation part of Scenario 3 which is *generating traces by using requirements relations and verification of architecture*, the three components above are used. First, the traces are generated in the output trace model 1

by the component *trace generator by using verification result*. Then the output trace model 1 is used as an input trace model by the component *trace generator by using requirements relations* to generate traces based on requirements relations in the input requirements model.

Trace Validator in ATL: The input of the component is the *Input Architecture Model*, the *Input Trace Model*, and the *Input Requirements Model*. The component is used in the trace validation part of all scenarios. It is implemented as an ATL transformation. The component checks the validity of assigned traces between R&A by using verification output or requirements relations. It can also check the validity of requirements relations by using traces between R&A. The output is the *Output Error Model* which contains invalid traces and invalid requirements relations.

Requirements Relation Generator using Traces in ATL: The input of the component is the *Input Architecture Model*, the *Input Trace Model*, and the *Input Requirements Model*. The component is used in the trace generation part of Scenario 4. It is implemented as an ATL model transformation. The component generates new requirements relations based on traces in the *Input Trace Model*. The output is given in the *Output Requirements Model* which contains only the generated requirements relations.

6.7.3 Tool Features

We describe the most important features of the tool: *verifying architecture*, *displaying generated traces*, and *displaying invalid traces*.

Verifying Architecture: We use the Open-Source AADL Tool Environment (OSATE) – Topcased [204] which includes an AADL front-end and architecture analysis capabilities as plug-ins. The plug-in [182] developed by Artur Boronat is used to generate Maude representation of AADL models which can be simulated and verified. Figure 6.17 shows the OSATE-Topcased with AADL-Maude plugin.

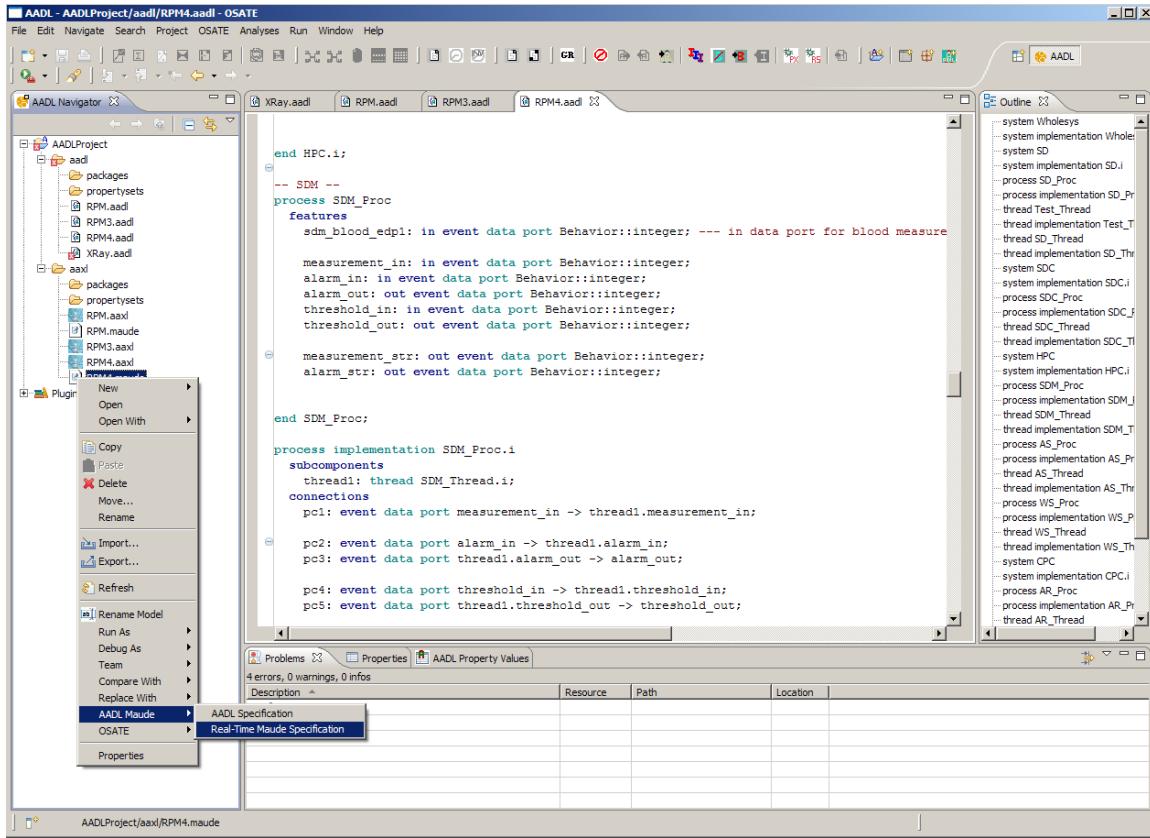


Figure 6.17 OSATE with AADL-Maude Plugin

In Maude, we can verify the software architecture for reformulated requirements in LTL. We use Eclipse plug-in developed in the context of MOMENT2 [30] to run Maude under Windows. Figure 6.18 gives the GUI for verifying architecture activity in Figure 6.15.

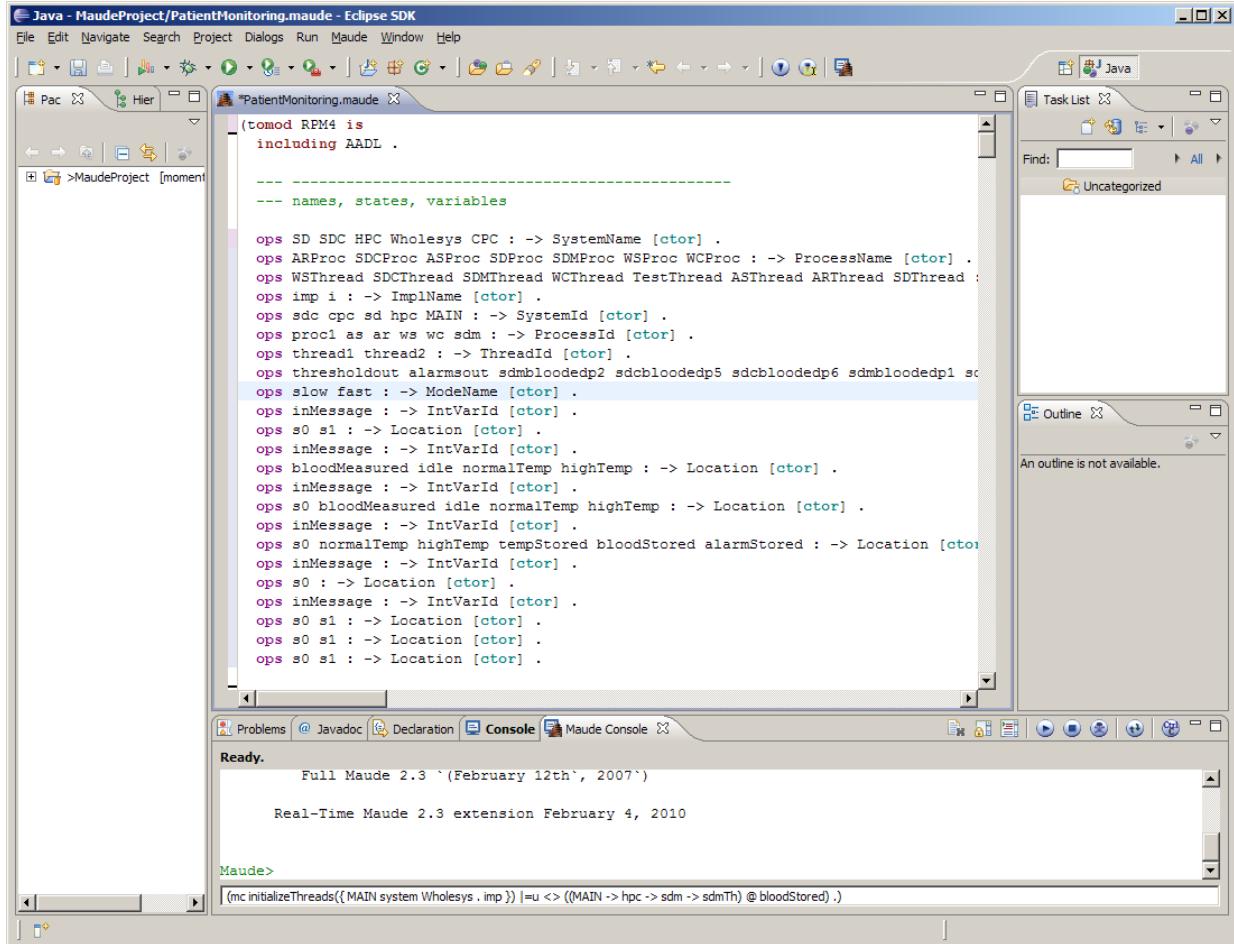


Figure 6.18 Maude Editor in Eclipse for Verifying Architecture

The window in Figure 6.18 displays the generated Maude code from AADL model. In the bottom of the window, the software architect can enter the LTL formula in order to verify the architecture.

Displaying Generated Traces: We use Eclipse model editor (see Figure 6.19) to display the *Output Trace Model* in Figure 6.2 which is the output of generating trace activity in Figure 6.15.

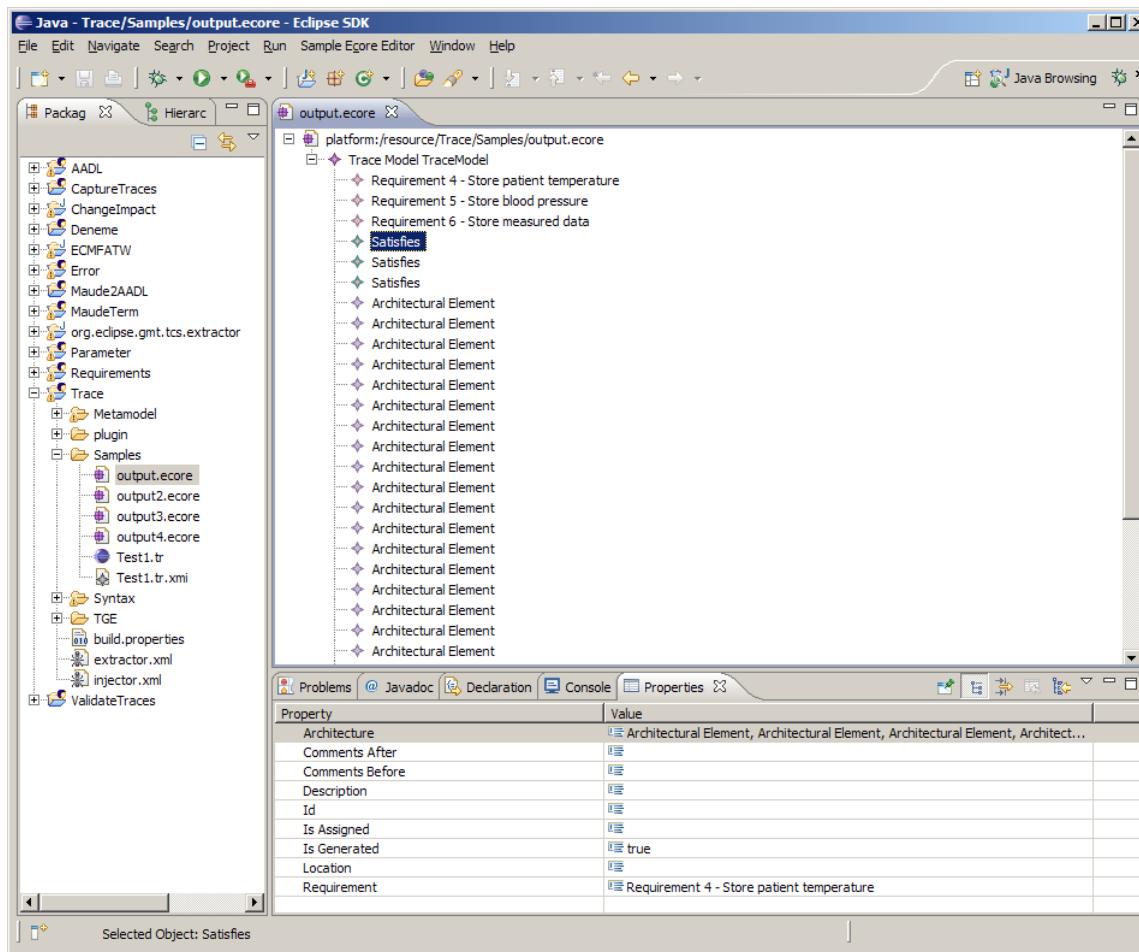


Figure 6.19 Output of the Generating Trace Activity

The right-hand side of the window shows the file *output.ecore* which is the output trace model. The trace model includes traces, requirements and architectural elements that are associated with these traces. The details of the chosen trace can be seen in the bottom of the window.

Displaying Invalid Traces: The output error model of validating trace activity in Figure 6.15 is displayed in Eclipse model editor. Figure 6.20 shows the output trace model in Eclipse model editor.

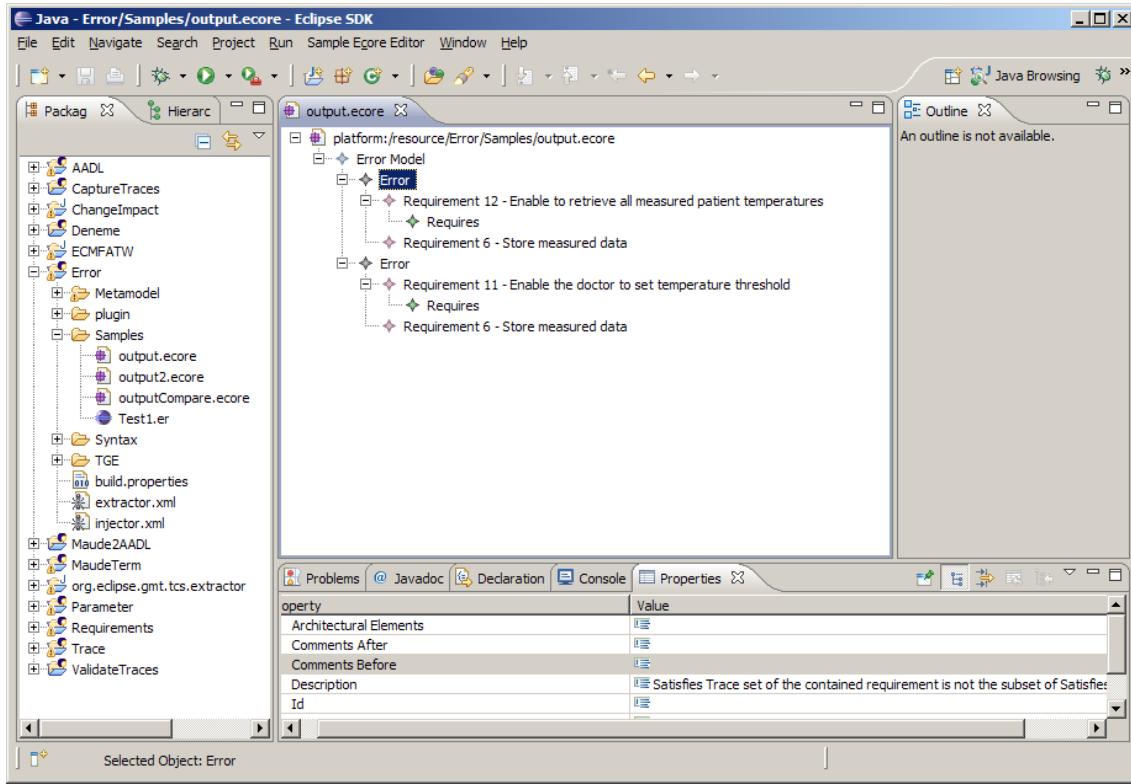


Figure 6.20 Output of the Validating Trace Activity

The right-hand side of the window shows the *Output Error Model*. The model contains requirements and requirements relations which cause the invalidity in the trace model. The architectural elements traced from the requirements in the error model can be reached in the trace model in Figure 6.19.

6.7.4 Evaluation of the Tool

Our tool can be evaluated regarding different qualities like *usability*, *performance* and *scalability*. The tool usability mainly depends on the usability of the Eclipse environment. For the counter example and execution traces (the output of the component *Architecture Verification in Maude*), no GUI is provided. For a prototype we consider this to be acceptable. In this section, we conduct performance and scalability tests of the tool for generating and validating traces. Our tool uses model checking techniques in verification of architecture for functional requirements. It is known that these techniques may have scalability and performance problems in handling large amounts of model elements and states. Therefore, we focus on model checking part of our tool in the performance and scalability tests.

Performance testing is conducted to evaluate the compliance of a system or component with specified performance requirements [1]. The requirement in our test is that the tool performs

in reasonable time (say less than one minute) with average number of architectural elements. We base our estimate for the average number of architectural elements on a report by McCormack et al. [174]. They characterize the differences in design structure between complex software products like Mozilla and Linux. The report shows that the architectural model of a real system contains around 2000 model elements. We take this finding as a base for our performance tests.

Scalability testing is a performance testing focused on ensuring the application under test gracefully handles increases in workload [1]. The workload in our performance test is the number of states. Our interpretation of scalability of the tool is the following: *the tool scales if the time spent by the tool increases linearly when the number of generated states increases linearly*.

Our dependent variable in the performance and scalability tests is the time for simulation and verification (in seconds). The independent variable used in the performance tests is number of elements in the architecture. We define the number of elements as follows: *number of component instances + number of feature instances + number of port connections where component, feature and port connections are the architectural elements in AADL*. The independent variable used in the scalability test is the number of states generated in the simulation. We define the number of states in the simulation as follows: *the number of states the simulation is enforced to explore*. These two variables are closely related to each other. If the number of elements is increased, it is likely that the number of states required for simulation and verification also increases. However, this does not always have to be the case. Assume that there are new architectural elements in the architecture for a new system property. New architectural elements may not increase the number of states in the simulation and verification of architecture for existing system properties.

Memory consumption is not measured in the performance tests. The runs for each performance test are executed six times. The runs are the cells in Table 6.2 and Table 6.3 for simulation times. The average for each run is derived from six executions. The performance tests are done with Intel(R) Core(TM)2 Quad CPU Q6600 running at 2.40 GHz with 4096 KB cache, and 2 GB of memory, running Kubuntu 10.04. We use Core Maude 2.4 for Linux. The models used in the performance tests are artificially created to test the tool with certain number of elements and states. The models used in the tests and the example AADL models given in this chapter do not have any real-time semantics. Real-time design and simulation are not the main focus of our approach. In the performance and scalability tests we use a version of operational semantics of AADL excluding the real-time semantics. The performance and scalability test results might be different with real-time semantics encoded in Real-Time Maude.

Performance test. The test is set up as follows. We increase the number of elements by adding components, data ports and data port connections to the architecture. We start with 2000 architectural elements and end up with 3000 architectural elements. The number of states for each run is 500, 1000 and 2000. The results of the performance test are shown in Table 6.2. Since the results of the performance test might be different when the verification result is an execution trace or a counter example, the performance test is done for both cases (see Table 6.2(a) and Table 6.2(b)). The standard deviation of the data is approximately 0.3%.

Table 6.2 Simulation Times in the Performance Test

# elements	Simulation Time (sec) for the Execution Trace		
	# states = 500	# states = 1000	# states = 2000
2000	7.8	15.9	33.8
2200	8.7	17.5	37.2
2400	9.3	19.4	40.4
2600	10.1	20.9	43.3
2800	10.9	22.4	46.5
3000	11.5	23.9	49.6

a) Simulation with Execution Trace

# elements	Simulation Time (sec) for the Counter Example		
	# states = 500	# states = 1000	# states = 2000
2000	2.6	5.2	10.8
2200	2.8	5.7	11.9
2400	3.1	6.3	13.0
2600	3.3	6.7	14.0
2800	3.5	7.2	15.2
3000	3.7	7.7	16.1

b) Simulation with Counter Example

According to these performance tests, the tool performs below one minute with average number of architectural elements in a real system. The increase in the simulation time is linear and up to 50 seconds for 2000 states (see Figure 6.21).

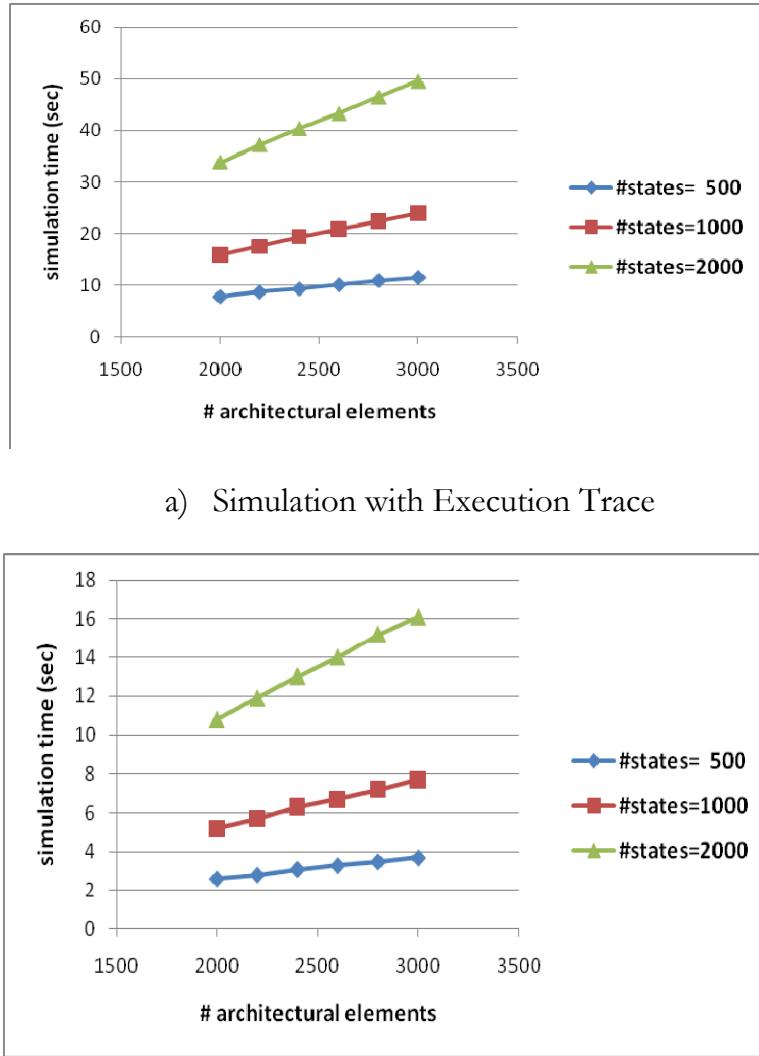


Figure 6.21 *Simulation Time as the Function of the Number of Architectural Elements*

Scalability test. The goal of this test is to investigate how the tool handles increases in the number of states over several orders of magnitude. Our independent variable is the number of states. We also compare the scalability test results of the tool using Maude with the results of the tool using different simulation and verification environments such as Alloy [126]. The same execution semantics of AADL in Maude is encoded in Alloy. The first part of the performance test is done in Maude with 10000 architectural elements (see Table 6.3(a)). Then, the second part of the performance test is done in Alloy (see Table 6.3(b)). In [163], we investigated simulation and verification in Alloy. We found that Alloy is not suitable for big number of model elements and states. Therefore, we choose to run the second part of

the performance test in Alloy with a smaller number of architectural elements (38 elements) (see Table 6.3(b)).

Table 6.3 Simulation Times in the Scalability Test

Number of States	Simulation Time (sec)
10	1.5
100	9.5
1000	82.1
3000	265.4
4500	401.8
5000	-

a) Simulation in Maude (# elements = 10000)

Number of States	Simulation Time (sec)
20	14.2
40	53.7
60	105.8
80	180.4
100	300.9

b) Simulation in Alloy (# elements = 38)

According to the scalability test results of our tool using Maude, the simulation time increases linearly when the number of states increases linearly (see Figure 6.22). We ran out of memory in Maude when we try simulation for 10000 architectural elements with 5000 states. For Alloy, the simulation time also increases linearly when the number of states increases, however, for much smaller number of architectural elements and much smaller number of states.

According to these test results, we conclude that our tool scales much better when using Maude rather than using Alloy.

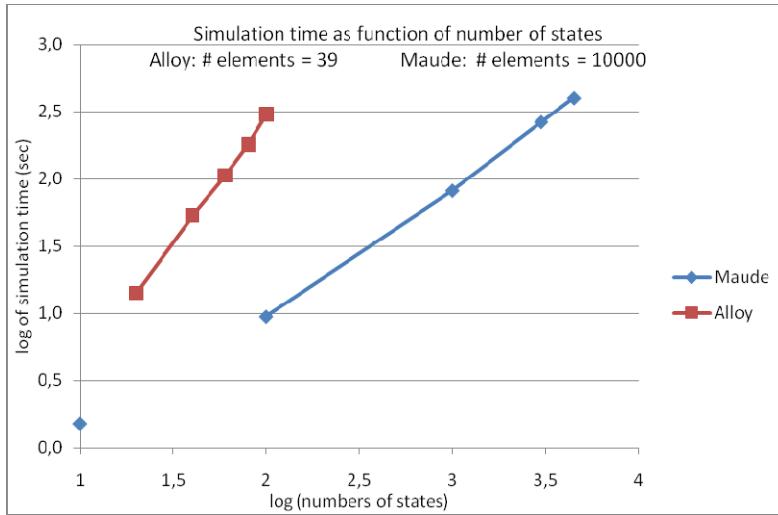


Figure 6.22 Simulation Time vs. Number of States in Alloy and Maude

We cover a subset of AADL semantics excluding real-time semantics in the tests. Our results are valid for this subset. The results depend on the modeling language and its semantics. The results may change with different AADL semantics or with a lower level design language like UML class and activity diagrams.

6.8 Discussion on the Approach

In our approach, the requirements are reformulated as formulas that encode properties of the software architecture. The requirement is first described as a formalized scenario, and then described as a *Linear Temporal Logic (LTL)* formula. The reformulation of the requirement is manual. There is no tool support or formal technique in our approach to ensure the consistency between the LTL formula and the requirement in natural text.

We use operational semantics of AADL formalized in Maude. The formal semantics for AADL in Maude is an interpretation of the informal and sometimes ambiguous descriptions in the AADL standard. We cover a subset of AADL semantics in our tool. As we already stated, our performance results for the tool in Section 6.7.4 are valid for this subset of AADL.

The tool uses AADL and Maude but the approach can be applied with another architecture description language and model checker. We can apply the simulation techniques in our approach to any other architecture description language which has operational semantics. The operational semantics of the architecture description language can be encoded and formalized in different environments such as GROOVE (GRaphs for Object-Oriented

VERification) [219] and Alloy [126]. Other logics like *Computation Tree Logic (CTL)* can be used in our approach.

A requirement may describe multiple system properties and/or a complex system property amenable to decomposition. In our approach it is not possible to explicitly state which property in the complex requirement fails. The requirements engineer should decompose the requirement into sub-parts (by using the *Contains* relation) until each requirement describes only one property which can be given as a single LTL formula.

The approach aims at preserving the requirements relations in their implementation in the architecture. There might be some cases where extra dependencies not identified in the requirements analysis are determined in the architecture. For instance, in the requirements analysis, the requirements engineer models two requirements as non-conflicting. In the implementation of the requirements in architectural design, the software architect might realize that these two requirements are conflicting with each other. The software architect should update the requirements model by introducing a *conflicts* relation between these two requirements.

6.9 Example for Trace Generation and Validation

In this section we give more examples for the Remote Patient Monitoring (RPM) system introduced in Section 6.5. It should be noted that the example is purely illustrative and can not be considered as a complete validation of the approach.

Section 6.9.1 illustrates reformulation of requirements and verification of architectures for the reformulated requirement in the example. Section 6.9.2 gives some generated traces in the example. In Section 6.9.3, we show some invalid traces identified in the example trace model.

6.9.1 Verification of Architecture for Functional Requirements

We verify the architecture of the RPM system for the following functional requirement.

Requirement 4 *The system shall store patient temperature measured by the sensor in the central storage.*

Requirement 4 is reformulated as a formalized scenario in terms of solution domain.

Formalized Scenario: *(contains(SD_TEMP_EDP1, DI)), (contains(SDM_TEMP_STRG, DI))*

The formalized scenario states that if the data instance *DI* is contained by the data port *SD_TEMP_EDP1* of Sensor 1 (*SD* component), then the result of the computation is that

the data instance DI is stored in the data store SDM_TEMP_STRG of the component SDM . The following is the LTL formula in Maude for the formalized scenario:

LTL Formula: $(mc \text{ initializeThreads}(\{ MAIN \text{ system Wholesys . imp } \}) |=u <> ((MAIN -> hpc -> sdm -> sdmTh) @ temperatureStored) .)$

The formula states that if the data instance DI is contained by the data port SD_TEMP_EDP1 of Sensor 1, then eventually in the future the state in the state transition system in the $sdmTh$ thread is set to the $temperatureStored$ state (the data instance DI is stored by the data store SDM_TEMP_STRG of the SDM component). When we check the formula on the architecture in Figure 6.6, the result is that the requirement is satisfied. Then, the *Satisfies* trace is generated for the requirement and the elements in the execution trace.

6.9.2 Generating Traces

In this section, we show some generated traces in the example. Consider the following requirement for the RPM system.

Requirement 5 *The system shall store patient blood pressure measured by the sensor in the central storage.*

In Section 6.9.1, we already showed that Requirement 4 is satisfied by the architecture. It can be shown that the architecture also satisfies Requirement 5. The *Satisfies* traces are generated for Requirement 4 and Requirement 5 accordingly (see Figure 6.23).

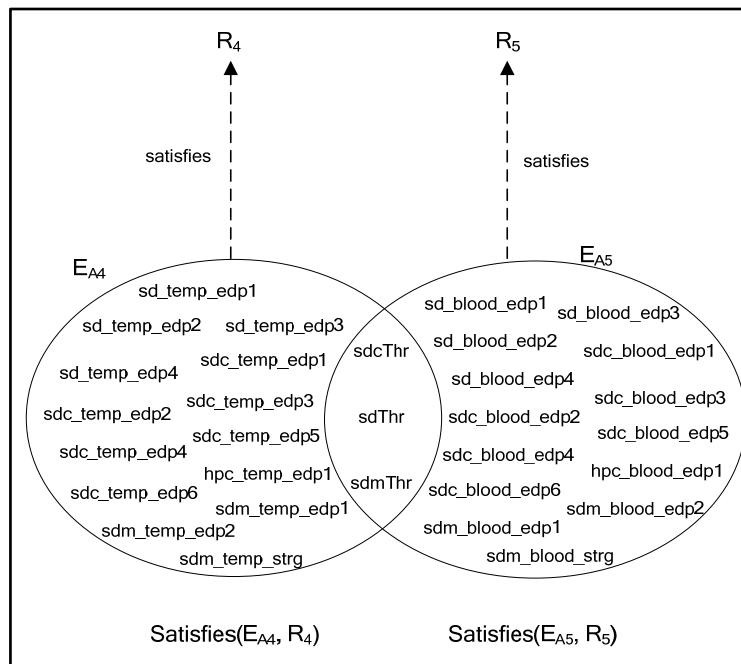


Figure 6.23 Generated ‘Satisfies’ Traces by Using Verification Results

The requirements model in Figure 6.5 states that Requirement 4 and Requirement 5 refine Requirement 6 which is the following:

Requirement 6 *The system shall store data measured by sensors in the central storage.*

Based on the constraints in Figure 6.9, the set of the generated *Satisfies* traces for Requirement 6 is the union of the trace sets for Requirement 4 and Requirement 5 (see Scenario 3). Figure 6.24 shows the generated *Satisfies* trace for Requirement 6 by using the requirements relations.

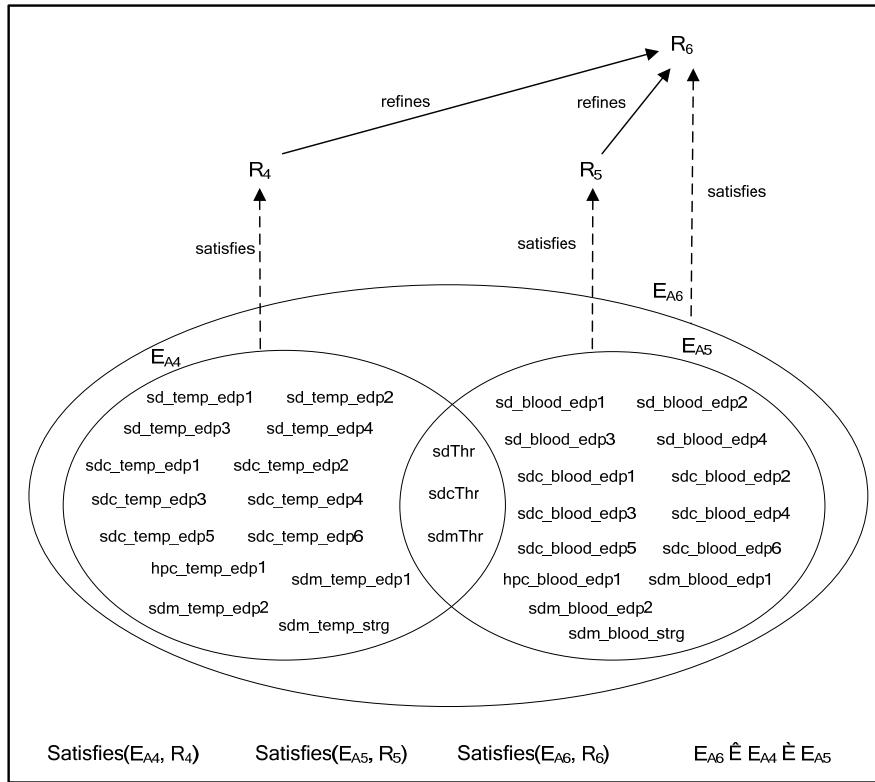


Figure 6.24 Generated ‘Satisfies’ Traces by Using Requirements Relations

Traces can also be used to generate requirements relations (see Scenario 4). Consider the following requirement for the RPM system.

Requirement 12 *The system shall enable the doctor to retrieve all stored temperature measurements for a patient.*

We already showed that Requirement 4 is satisfied by the architecture. It can be shown that the architecture also satisfies Requirement 12. The *Satisfies* trace is generated for Requirement 12 accordingly.

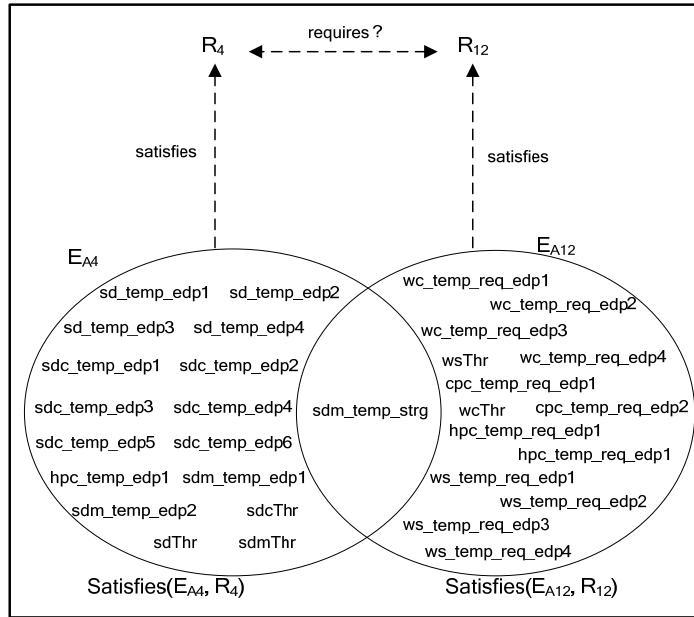


Figure 6.25 Generated Requirements Relation by Using Traces

Figure 6.25 shows the intersection of traces for Requirement 4 and Requirement 12. Based on the constraints in Figure 6.9, there might be a *Requires* relation between Requirement 4 and Requirement 12 if the intersection of traces is non-empty. The output of the tool is only the candidate requirements relations. For the *Requires* relation, the tool can not suggest the direction of the relations. The final decision about the relation should be made by the architect. We analyzed the suggested relation and concluded that Requirement 12 *requires* Requirement 4.

6.9.3 Validating Traces

In this section, we perform validation of traces in the example. Section 6.9.2 showed the generated *Satisfies* traces for Requirement 6. There are also assigned *AllocatedTo* traces for the same requirement. Figure 6.26 gives the generated *Satisfies* and assigned *AllocatedTo* traces for Requirement 6.

The traces in Figure 6.26 are validated according to the differences of the architectural element sets of the traces in the Venn diagram in Figure 6.12 (see Scenario 2). We check the software architecture for Requirement 6. Although Requirement 6 is allocated to the components *AS* and *CPC_AR*, these components are not involved in the architecture design for Requirement 6. We concluded that two *AllocatedTo* traces to the components *AS* and *CPC_AR* are false positive traces.

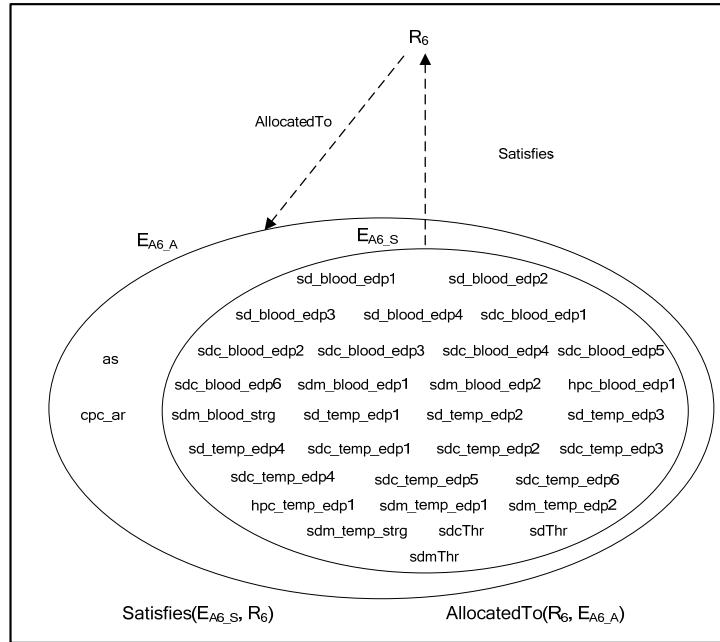


Figure 6.26 Generated ‘Satisfies’ and Assigned ‘AllocatedTo’ Traces for Requirement 6

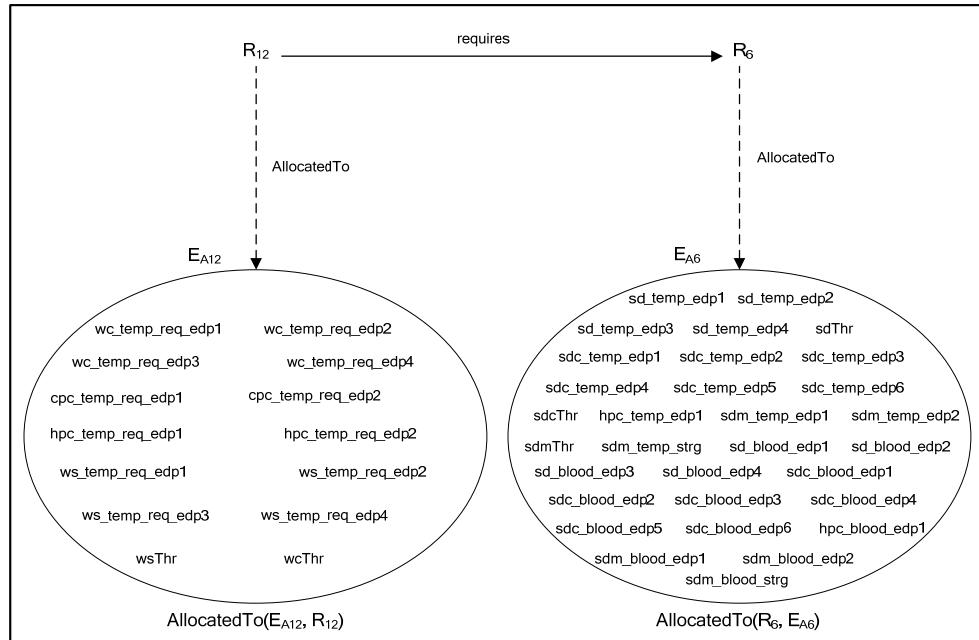


Figure 6.27 Assigned ‘AllocatedTo’ Traces with Requirements Relation

Figure 6.27 shows the assigned *AllocatedTo* traces for Requirement 12 and Requirement 6 with the *Requires* relation. The *Requires* relation between Requirement 12 and Requirement 6 is a given relation. The traces in Figure 6.27 are validated with the *Requires* relation between

Requirement 12 and Requirement 6 (see Scenario 1). Based on the constraints in Figure 6.9, the sets of the assigned *AllocatedTo* traces for Requirement 12 and Requirement 6 should have a non-empty intersection. We re-inspected Requirement 12, Requirement 6 and the architecture. We concluded that some traces are missing for Requirement 12. We allocated Requirement 12 to the architectural element *SDM_TEMP_STRG* to which Requirement 6 is also allocated.

For the traces and requirements relation in Figure 6.27, we decided that the *Requires* relation is valid and the traces should be corrected. However, there might be cases where the requirements relation is identified as invalid based on the constraints in Figure 6.9 (see Scenario 4). Figure 6.28 shows the assigned *AllocatedTo* traces for Requirement 10 and Requirement 6 with the *Refines* relation. Requirement 6 and Requirement 10 are the following:

Requirement 6 *The system shall store data measured by sensors in the central storage.*

Requirement 10 *The system shall store all generated temperature alarms in a central database.*

Based on the constraints and by analyzing requirements, we concluded that the *Refines* relation between Requirement 10 and Requirement 6 is invalid.

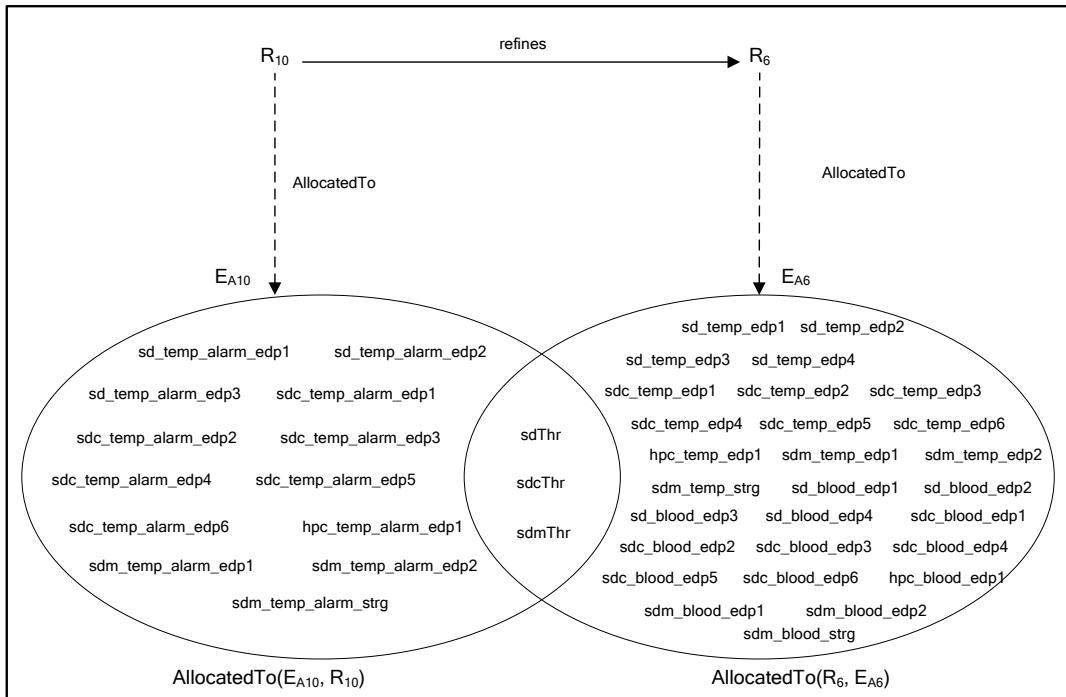


Figure 6.28 Assigned ‘AllocatedTo’ Traces with an Invalid Requirements Relation

When we delete the invalid given relations, some of the inferred relations might be automatically deleted. Figure 6.29 gives some given and inferred requirements relations for Requirement 10.

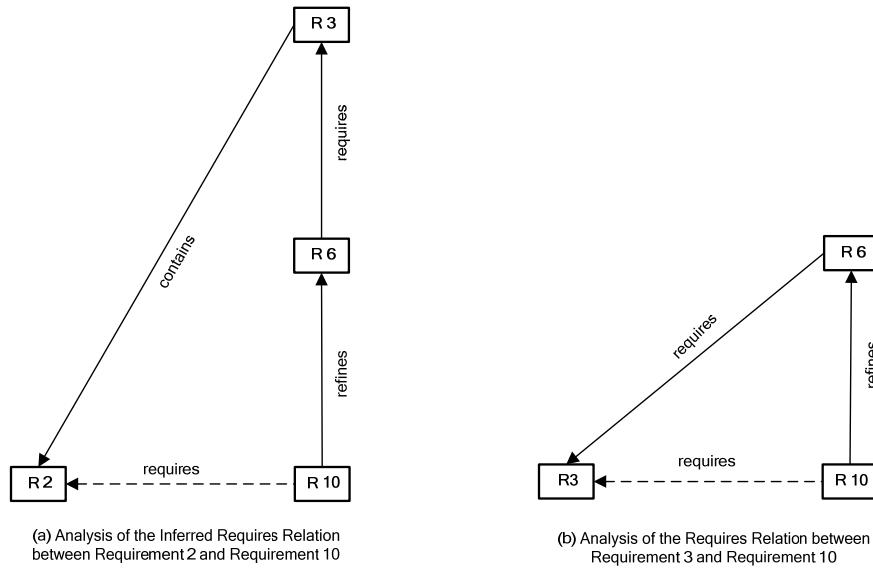


Figure 6.29 Given and Inferred Requirements Relations for Requirement 10

Requirement 2 *The system shall measure blood pressure from a patient.*

Requirement 3 *The system shall measure blood pressure and temperature from a patient.*

Requirement 6 *The system shall store data measured by sensors in the central storage.*

Requirement 10 *The system shall store all generated temperature alarms in a central database.*

The solid arrows indicate the given relations; the dashed arrows denote the relations inferred from the given relations. The *requires* between Requirement 10 and Requirement 2 is inferred from the *refines* between Requirement 10 and Requirement 6, *requires* between Requirement 6 and Requirement 3, and *contains* between Requirement 2 and Requirement 3 (see Figure 6.29(a)). The *requires* between Requirement 3 and Requirement 10 is inferred from the *refines* between Requirement 10 and Requirement 6, and *requires* between Requirement 3 and Requirement 6 (see Figure 6.29(b)). Removal of the *Refines* relation between Requirement 10 and Requirement 6 automatically removes the inferred *requires* relations in Figure 6.29(a) and (b).

For the validation of traces and requirements relations for cases like in Figure 6.27 and Figure 6.28, our tool gives the traces and requirements relations which do not obey the

constraints. The architect should decide about either the traces or the requirements relations are invalid.

6.10 Related Work

We discuss related work in six categories: *Types and Semantics of Traces*, *Generating and Validating Traces*, *Conformance Assessment*, *Architecture Analysis*, *Analyzing AADL Models* and *Tool Support*.

6.10.1 Types and Semantics of Traces

A number of approaches address types and semantics of traces between R&A. Paige et al. [206] focus on how to define traces with tool-supported semantics. According to [206] semantically rich traces possess three characteristics: (1) traces are typed, (2) traces conform to a case-specific trace metamodel, and (3) the case-specific metamodel should be accompanied by a set of case-specific constraints, which cannot be captured by the metamodel. The trace metamodel in Section 6.3, which is a case-specific trace metamodel, contains the *Satisfies* and *AllocatedTo* traces with semantics in FOL. Based on the semantics we can generate and validate traces between R&A in a formal manner. The trace metamodel includes the case-specific trace information such that the trace is generated or assigned. This type of trace information can prevent users and tools from establishing illegitimate traces [206]. One of the case-specific constraints, not captured by the metamodel, is that there cannot be both generated *Satisfies* and *AllocatedTo* traces for the same requirement.

Aizenbud-Reshef et al. [6] state the need for semantics of traces in general. They present an approach to defining operational semantics for traces in UML. The semantic property of a trace is a triplet (*event*, *condition*, and *actions*). This triplet is very much dependent on change impact analysis. Therefore, it is hard to use the semantics in [6] for other purposes like generating and validating traces.

Ramesh and Jarke [215] define traces between R&A: *allocated to* and *satisfy* which have similar definitions with trace types in this chapter. Khan et al. [138] define a dependency model to analyze the impact of evolving requirements dependencies and architecture changes. The dependency model consists of six types of traces: *goal dependency*, *service dependency*, *conditional dependency*, *temporal dependency*, *task dependency* and *infrastructure dependency*. Lago et al. [147] propose following trace types between feature models (requirements) and structural models (architecture): *drive*, *modify*, *depend-on*, and *influence*. There is no formal semantics of the trace types in [138] [147] [215]. All these types can be mapped to our trace types.

6.10.2 Generating and Validating Traces

A number of approaches provide generating and validating traces. Egyed et al. [72] [71] [70] provides an automated traceability approach that uses a small number of traces as input. In [72] [70], the source code is executed according to some scenarios and then traces are generated between requirements and source code. Footprint graph is used to detect the incomplete and incorrect input. Dependencies between requirements can be detected based on overlaps among the lines of code implementing those requirements. There is no formal semantics of trace types (*hypothesized, generated, validated* and *observed* traces) in [72] [70]. Similarly to his work, we use reformulation of requirements as scenarios.

Schwarz et al. [228] describe a graph-based traceability approach. Generation and maintenance of traces are handled by model transformations. The *Satisfies* trace is provided without any formal semantics or textual definition. Components, interfaces and ports in the architecture are created automatically from requirements and use cases by using heuristics. Our work assumes that architecture is created manually.

Information retrieval methods are proposed for trace generation. Antoniol et al. [13] propose an approach for recovering traces between source code and documentation (mainly requirements specification) using information retrieval methods. Hayes et al. [109] introduce another approach for trace generation. The assumption of these works is that programmers use meaningful names for program items so that the analysis of the mnemonics can help to associate high-level concepts with source code.

Grechanik et al. [102] support generating traces between types and variables in Java programs and elements of use-case diagrams (UCD). The approach combines program analysis, run-time monitoring, and machine learning to generate traces. Relations between program entities are compared with corresponding relations between elements in UCDs only to validate traces. Cysneiros and Zisman [58] describe an approach to support traceability for agent systems. Although it is claimed that the approach uses six types of traces and semantics of these types, no semantics for the trace types is provided in [58]. The approach supports generating traces between design models and code specification by checking synonyms. Instead of checking synonyms our approach uses semantics of traces and requirements relations. Bonde et al. [26] describe an interoperability approach based on generating a trace model by using model transformations. This work focuses on traces between platform independent and platform specific models in MDA context.

Mader et al. [167] address modification and enhancement of existing traces after changes to artifacts. The approach does not support trace generation. On the contrary, in our approach initial traces can be generated by using architecture verification techniques. A Visual

Traceability Modeling Language (VTML) is proposed by Mader et al. [166]. VTML allows users to model trace queries by hiding underlying technical details. The queries created with VTML can be applied on traces generated and validated by our approach.

6.10.3 Conformance Assessment

Conformance assessment is the act of checking whether a requirement is satisfied [11]. The assessment can be testing, inspection, model checking or conformation transformation usage (see [10] for conformation transformation usage). The usage of architecture verification with requirements relations in our approach can be considered as a conformance assessment of properties in the requirements and architecture.

Almeida et al. [11] propose a framework that supports management of traces between requirements and design. The framework provides a notion of conformance between application models which reduces the effort for conformance assessment. The conformance between various application models at different levels of abstraction is assessed. In our approach, we focus on conformance assessment between requirements and software architecture. We do not consider the case where there are multiple design models at various abstraction levels.

Paige et al. [207] give a definition of refinement between models via consistency checking. Formal definitions for model consistency are provided with the definition of refinement in MDA. The consistency of platform specific and independent models is ensured with cross-model rules which actually check the preservation of properties between two models. There are other conformance assessment approaches by Egyed [69], Abi-Antoun et al. [3] [4], Moriconi et al. [184], Heckel et al. [112] and Oquendo [205]. Most of these works focuses on the conformance assessment for architecture and detailed design. Our approach does conformance assessment for requirements and architecture design.

6.10.4 Architecture Analysis

Simulation and model checking of software architecture are parts of our approach for trace generation and validation. We studied the literature about behavioral and static analysis of software architecture models. Zhange et al. [264] give a classification and comparison of model checking software architecture techniques. According to the survey in [264], CHARMY [209], using the SPIN model-checker, is one of the most recent architecture analysis approaches. A similar approach that uses SPIN for verifying software architecture is proposed by Bose [31]. The works in [209] and [31] use UML-based notations instead of architecture description languages.

According to [264], Wright language proposed by Allen and Garlan [9] can be considered as the first work on model checking techniques for software architecture. The extension of the work in [8] addresses the problem of specifying and analyzing dynamic behaviour of architectures. Dynamic behavior is distinguished from the steady-state behavior where the computation performed by a system without reconfiguration [8]. Magee and Kramer [170] outline examples of language features for dynamic structure. There are approaches for analyzing dynamic behaviour: Darwin [171] [168], Chemical Abstract Machine (CHAM) [56] [57] [125], dynamic architecture verification using DynAlloy [39] [38], reconfiguration analysis in service oriented architectures [15], and behaviour preservation in dynamic architectures [112]. Our approach does not support the analysis of dynamic behaviour.

There are works [64] [73] [187] about extending architectural description languages with statechart semantics to analyze the internal component behavior. These works are similar to the behavioral annex for thread and subprogram behavior in AADL. In behavioral annex, the behavior of a thread is modeled as a set of states with pre and post conditions. Ölveczky et al. [197] [198] implement pre and post conditions in Maude rewriting rules.

Boudiaf et al. [32] use rewriting rules in Maude to give the behavioral semantics of multi-agent system models for architecture analysis. The difference with the architecture analysis we use is that Boudiaf et al. perform architecture analysis on multi-agent system models. ArchJava [7] is an extension to Java that unifies an architecture with its implementation. It is possible to check if architectural properties are preserved in source code. On the contrary, we do not couple source code with architecture.

Apart from behavioral analysis, there are static analysis techniques for architecture verification. Allen and Garlan [9] use the static analysis tool FDR [84] to check deadlock and component-connector compatibility. Naumovich et al. [188] use static analysis tools based on flow equations. One of the drawbacks of using static analysis is that some dynamic features of architecture description languages might cause difficulties for static analysis.

6.10.5 Analyzing AADL Models

In the previous subsection, we give the literature about general architecture analysis techniques. There are also works particularly studying architecture analysis in AADL models. Delanote et al. [63] explore the use of AADL in model driven development. However, the authors do not adapt any architecture analysis technique to AADL models.

Berthomieu et al. [20] [21] give an approach for formal verification of AADL specifications. A subset of AADL is translated into an extension of Petri nets called *Fiacre* language [21]. Chkouri et al. [46] propose another analysis approach using translation between AADL and

BIP (Behavior Interaction Priority) language. The analysis technique [197] we use gives a formal executable semantics to an AADL model with a behavior annex specification of its thread behavior. On the contrary, the approaches in [20] [21] [46] use translations into imperative languages. Similar to [197], Yang et al. [263] propose a formal semantics in Timed Abstract State Machine (TASM) for a limited set of AADL behavior annex (periodic threads and no modes).

Jahier et al. [127] provides an AADL analysis approach in which the behavior of software components are developed as AADL subprogram execution. In [127] the testing tool Lurette [128] is used for simulation; verification of architecture is done by the Lesar model-checker [217]. Abdoul et al. [2] propose an AADL model transformation which provides a formal model for model checking activities. Hugues et al. [116] present a tool suite for analyzing AADL models. In [116], it is considered that subprograms in AADL encapsulate the behavior of architecture. Similar to [197], Benammar et al. [18] [19] propose the use of rewriting logic in Maude as a formalism for modeling behavior in an AADL architectural description. On the contrary, in [18] [19], the behavior of a thread is specified directly in Maude.

Varona-Gomez et al. [253] translate AADL models to SystemC models for performance analysis. Bozzano et al. [34] [33] present an AADL analysis approach which supports Error Model Annex for modeling faults and repairs. Li et al. [155] propose the use of Communicating Sequential Processes (CSP) for simulation of AADL models. The works in [104] and [232] focus on analyzing schedulability with a behavior of a subset of AADL. All these approaches assume that the thread behavior is specified outside AADL.

Apart from simulating and verifying AADL models, de Niz et al. [61] propose the use of AADL models to analyze potentially unintended system behavior. Gilles and Hugues [91] [92] present a domain specific language (REAL – Requirement Enforcement Analysis Language) for AADL. Contrary to our approach, the approach in [91] [92] does not focus on simulation and verification of AADL models.

6.10.6 Tool Support

Some requirements management tools support traces from requirements to system implementation. The INCOSE management tool survey [124] evaluates these tools according to the criterion *traceability analysis*, that is, what kind of trace links the tools provide and what kind of analysis is performed by the tools. According to the responses of tool vendors in the survey, current industrial tools mostly provide tracing requirements to system implementation such as software architecture with integration of other modeling tools.

However, they do not provide mechanisms of trace generation and validation for requirements and architecture.

IBM Rational RequisitePro [119] provides only two types of trace between requirements, requirements & design, and requirements & implementation: *traceFrom* and *traceTo*. These two trace types indicate only the direction. IBM Telelogic Doors [120] provides a mechanism of describing functional decomposition and analysis in UML. The tool supports two types of trace: *internal* and *external*. Internal traces can be created between any two elements in the same model such as requirements relations, while external links can be used to link elements in different models such as traces between R&A. The requirements engineer can also specify his own trace type. Borland Caliber [27] provides only one trace type. This type can be used for different purposes such as part-whole and refinement. A trace can be established between any two artifacts. These artifacts can be of the same type or different types and even external artifacts, like files, UML elements or test cases. The reasoning facilities of the tools IBM Rational RequisitePro, IBM Telelogic Doors, and Borland Caliber are based only on the transitivity property of the traces. These tools do not support validation of traces.

In TopTeam Analyst [246], there are four trace types. Three of these traces (*traces into*, *impact*, *used in*) are directed and one of them (*trace*) is undirected. This undirected trace is considered as a generic trace type for other trace types. None of the trace types have formal semantics. The tool does not support trace generation and validation.

6.11 Conclusions

In this chapter, we focused on traces between requirements and architecture. Trace types with formal semantics were proposed. The formalization of traces is based on the idea that the properties stated by requirements (problem domain) are satisfied in the architecture (solution domain). These properties can be reformulated in terms of the architectural solution and verified. The prerequisite for the verification is the presence of a formal executable specification of the dynamic semantics of the architecture description language.

Our approach uses Maude, a formal language based on equational and rewriting logic, and MDE technologies such as Eclipse EMF and ATL. The architecture is modeled in Architecture Analysis and Design Language (AADL). Maude is used for simulating and verifying software architecture. Model transformations in ATL are used to generate and validate traces by using verification results and requirements relations.

In this chapter, we answered *Research Question 4* (*How to model requirements, software architecture and traces with their semantics for change management?*) and *Research Question 5* (*How can we formally check if the evolved architecture satisfies evolved requirements? How can we become sure that traces are up-to-date?*) raised in Chapter 1. The entities *Trace*, *Satisfies* and *AllocatedTo* in the trace metamodel are the aspects of traces to be modeled. These entities with their semantics are used to reason about traces. By using architecture verification techniques in our approach it is checked if the evolved architecture satisfies evolved requirements. Trace generation and validation are used to keep traces up-to-date and also to automatically generate initial traces.

There are some open issues in the approach. The approach requires the adaptation of the output of requirements engineering activities for software verification. For large software development companies there are challenges in this adaptation [224]. Reformulation of requirements in terms of solution domain is one of these challenges. It is a part of a design process and is hard to automate. The architect might still need to check the generated traces. In case of false positives the requirements model and relations should be checked. Therefore, we suggest an iterative semiautomatic process of applying our approach. In such a process, the software architect can gradually improve the quality of the traces and the requirements. Case studies conducted with the industry [47] shows that it is hard to reformulate requirements as LTL/CTL formulas. Domain-specific languages can be used for requirements of certain type that allow compilation of LTL/CTL formulas [47]. Starting from natural language, Semantics Business Vocabulary and Rules (SBVR) [202] can support reformulation of requirements in terms of LTL/CTL formulas. Extending our approach with this kind of languages will ease the reformulation of requirements.

We mainly focused on scalability issues in our tool for generating and validating traces. Since model checking techniques may have problems in handling large amounts of model elements and states, the scalability of our tool depend on the scalability of the model checking algorithms in Maude. Our tool needs further improvement for usability. The core parts of the tool are implemented. However, integration of these parts is currently done manually and we need a user interface to control all these parts.

In Chapter 5, we presented an approach for using requirements relations and their semantics for change impact analysis. In this chapter, we defined traces between requirements and architecture models. Chapter 7 applies semantics of traces and requirements relations to change impact analysis for software architecture.

Chapter 7

7 Change Impact Analysis in Software Architecture

In this chapter, we aim at improving change impact analysis in software architecture models by using architecture verification techniques and formal semantics of traces. Our technique has two parts that use the approaches in Chapters 5 and 6. The first part is to identify the architectural elements that implement the system properties related to proposed requirements changes. We extended TRIC for determining candidate impacted architectural elements. The second part is to propose possible changes for software architecture when the software architecture does not satisfy new and/or changed requirements. The technique is based on architecture verification. The output of verification is a counter example if the requirements are not satisfied. The counter example is used together with a classification of architectural changes in order to propose changes in the software architecture. The technique supports the architect to change the architecture in order to satisfy the requirements.

7.1 Introduction

Chapter 5 presented a change impact analysis approach in requirements models based on the formal semantics of requirements relations. In Chapter 6 we presented an approach that provides trace establishment by using semantics of traces between Requirements (R) and Architecture (A) (see Figure 7.1 for the *Satisfies* and *AllocatedTo* traces).

Once the requirements engineer analyzes the impact of a change in requirements, the software architect needs to identify the impact of this change in software architecture. By using only the transitive closure of requirements relations and traces between R&A, the software architect may conclude that all architectural elements in the architecture are impacted. Without any additional semantic information about the requirements relations,

traces and change, he/she may have to analyze the whole software architecture for a single change. Furthermore, without considering semantics, change impact analysis may produce high number of false positive impacts. Consequently, the cost of implementing a change may become several times higher than expected. For example, in Figure 7.1 a change proposed for R_n is propagated to R_3 by using the semantics of the *contains* relation. For the proposed change in R_n , the architectural elements C_3 , C_4 , C_5 and C_6 can be traced from R_n , or C_4 and C_5 can be traced from R_3 . C_3 and C_6 are also candidate impacted if we start tracing from R_n although they are not related to the changed property in R_n . In addition to C_4 and C_5 , the software architect has to inspect C_3 and C_6 to identify what to change in the architecture.

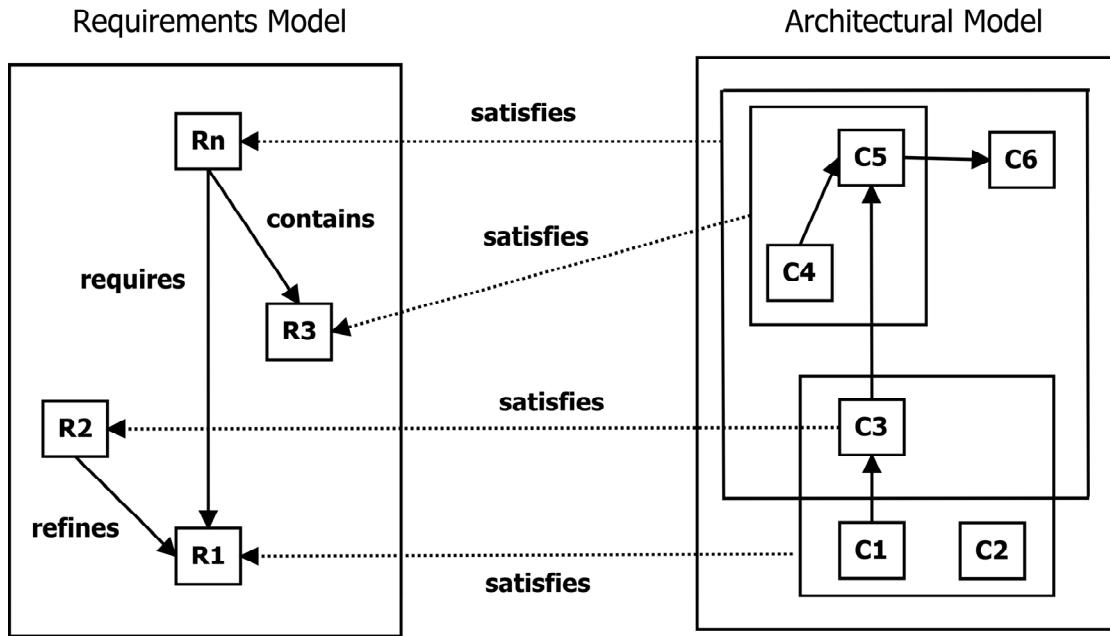


Figure 7.1 Within-Model and Between-Model Traces with Requirements Relation Types and Trace Types between Requirements and Software Architectures

In this chapter we present a change impact analysis technique for software architecture using architecture verification and semantics of traces. Our technique has two parts. The first part is to identify the architectural elements that implement the system properties to which proposed requirements changes are introduced. Semantics of requirements relations and traces is used in the first part. We extended TRIC for determining candidate impacted architectural elements. The software architect starts changing the software architecture based on the candidate impacted parts of the architecture. After the changes are implemented, the software architecture may not satisfy the new/changed requirements. The second part of our technique is to propose possible architectural changes when the software architecture does not satisfy the new and/or changed requirements. The technique is based on architecture

verification. The output of verification is a counter example if the requirements are not satisfied. The counter example is used together with a classification of architectural changes in order to propose changes in the software architecture. The technique is semi-automatic and iterative.

In this chapter, we answer *Research Question 4* (*How to model requirements, software architecture and traces with their semantics for change management?*) and *Research Question 5* (*How can be a change in a requirement propagated to other requirements and to software architecture? How can we support the requirements engineer and software architect for performing changes? How can we formally check if the evolved architecture satisfies evolved requirements?*) raised in Chapter 1. With the approach for change impact analysis in requirements models we address the issues about propagation of changes from a requirement to architectural elements.

This chapter is structured as follows. Section 7.2 describes the approach. Section 7.3 presents the first part of the approach, identifying candidate impacted architectural elements. In Section 7.4, we describe proposing architectural changes. Section 7.5 explains the tool support. Section 7.6 describes the related work, and Section 7.7 concludes the chapter.

7.2 Approach

We aim at identifying impacted architectural elements when a requirement is changed. We rely on previously defined requirements and trace metamodels. In addition, in this chapter we develop the following:

- **Identifying candidate impacted architectural elements.** We identify which parts of the architecture are impacted by a proposed change in requirements (Section 7.3).
- **Proposing architectural changes.** We propose possible changes for software architecture when the software architecture does not satisfy the new/changed requirements (Section 7.4).

We provide tool support and illustrate the feasibility of our approach in an example.

- **Tool support.** We describe the design and implementation of a prototype tool for identifying impacted architectural elements and proposing architectural changes (Section 7.5).
- **Example.** The approach is illustrated with an example. The example is the Remote Patient Monitoring (RPM) system which is also used in Chapter 6. Part of the RPM requirements document is given in Appendix F.

7.3 Identifying Candidate Impacted Architectural Elements

The approach in Chapter 5 enables the requirements engineer to propose a change for a requirement and propagate the proposed change to related requirements. The output is the set of proposed changes for requirements with a propagation path in the requirements model. Our technique in this section focuses on determining the architectural elements that implement the system properties described by the requirements to which changes are proposed. We are concerned with the domain changes for requirements (see Chapter 5). By using formal semantics of requirements relations and traces between R&A, we identify which parts of software architecture are impacted by a proposed change in requirements. The impact is calculated by a change impact function. The change impact function takes a change type, a requirement to which the change is introduced, a set of requirements relations for the requirement and a set of all traces between R&A as input. The output of the change impact function is a set of architectural elements which are candidate impacted for the change in the requirement. The following is the signature of the change impact function.

$impact : SCT \times SR \times SSRR \times SST \rightarrow SSAE$

where SCT is the set of change types, SR is the set of requirements, $SSRR$ is the set of sets of requirements relations, SST is the set of sets of traces and $SSAE$ is the set of sets of architectural elements which are candidate impacted for the requirements change.

Traces in SST can be either generated *Satisfies* traces or assigned *AllocatedTo* traces. Given the domain changes that can be made to the requirements model, we describe rules to determine the impact of each requirements change type in software architecture (see Chapter 5 for *requirements change classification* and *semantics of changes*). The algorithm of the change impact function is based on the types of requirements changes. According to the type of requirements change, the algorithm may traverse the propagation path of the requirements change in the requirements model. Then, candidate impacted architectural elements are identified by using traces between requirements and architectural elements. The algorithm of the change impact function is given in Appendix I. In the following we give the main parts of the algorithm in pseudo code:

```

1   impact(ChangeType c, Requirement r, Set srl, Set st): Set {
2       Set sae = empty-set
3       If ((c is 'Add a New Requirements Relation') OR
4           (c is 'Delete Requirements Relation') OR
5           (c is 'Update Requirements Relation')) {
6               Return empty-set

```

```

7      }
8
9      If (c is 'Add Property to Requirement') { Return empty-set }
10
11     If (c is 'Add a New Requirement') {
12         If (srl is empty-set) { Return empty-set }
13
14         sae = getCandidateImpacts(c, r, srl)
15         Return sae
16     }
17
18     srlp = getRelationsInPropagation(c, r, srl)
19     sae = traversePropagationPath(c, r, srlp, st)
20     Return sae
21 } //End of impact function

```

Candidate impacted architectural elements are identified based on the type of requirements change. The algorithm checks the type of requirements change (see lines 3, 4, 5, 9 and 11).

- **Candidate Impacts for ‘Add a New Requirements Relation’, ‘Delete Requirements Relation’, and ‘Update Requirements Relation’.** If the change is ‘Add a New Requirements Relation’, ‘Delete Requirements Relation’ or ‘Update Requirements Relation’ (see lines 3 - 7), there is no impact on architecture. These change types improve the structure of the model without modifying overall system properties (see *refactoring* in Chapter 5). They have no impact on software architecture. However, trace constraints given in Chapter 6 should be checked after the changes.
- **Candidate Impacts for ‘Add Property to Requirement’.** If the change is ‘Add Property to Requirement’ (see line 9), there is no suggestion for candidate impacted architectural elements. If the added property is a new system property (see *domain changes* in Chapter 5), architectural elements that satisfy the existing properties related to the added property are candidate impacted. In the requirement itself, there is no explicit dependency between the existing properties and the added property. Therefore, it is not possible to automatically identify architectural elements that satisfy the existing properties related to the new system property as candidate impacted. The added property may just be an existing property added to the requirement to improve the structure of the model without modifying overall system

properties (see *refactoring* in Chapter 5). There is no impact on software architecture if the added property is not a new system property. The approach can not identify automatically if the added property is a new system property or not. There is no suggestion for candidate impacted architectural elements. The possible impact needs to be analyzed by the architect.

- **Candidate Impacts for ‘Add Requirement’.** If the change is ‘Add a New Requirement’ (see lines 11 - 16), either architectural elements traced from directly related requirements are candidate impacted or there is no impact (see Section 7.3.1).
- **Candidate Impacts for Other Changes.** If the change is none of the changes above (see line 19), the propagation path of the change is traversed to identify candidate impacted architectural elements (see Section 7.3.2).

The software architect takes design decisions to implement the change. Some or all of the architectural elements identified as candidate impacted may not be actually impacted because of the design decisions taken by the architect. New architectural elements might be introduced to the software architecture instead of changing the existing elements.

7.3.1 Candidate Impacts for ‘Add Requirement’

If the added requirement introduces a new system property (see *domain changes* in Chapter 5), architectural elements that satisfy requirements directly related to the added requirement are candidate impacted. If there is no new system property introduced by the change (see *refactoring* in Chapter 5), there is no impact on software architecture. Table 7.1 gives the change impact rules for the change type ‘Add Requirement’. Each cell gives the candidate impacted architectural elements for the change type in the row and the relations in the columns.

Table 7.1 Change Impact Rules for the Change Type “Add Requirement”

Change	Requirements Relation Types							
	R _i contains R _x	R _i refines R _x	R _i partially refines R _x	R _i requires R _x	R _x contains R _i	R _x refines R _i	R _x partially refines R _i	R _x requires R _i
Add R _x	No candidate impacted AE ⁵	No candidate impacted AE	No candidate impacted AE	AEs traced from R _i are candidate impacted	No candidate impacted AE	AEs traced from R _i are candidate impacted	AEs traced from R _i are candidate impacted	AEs traced from R _i are candidate impacted

⁵ ‘AE’ stands for ‘Architectural Element’

The requirement R_i in Table 7.1 denotes an existing requirement. R_x is the added requirement. The change ‘Add Requirement’ is not a domain change if $(R_i \text{ contains } R_x)$, $(R_i \text{ refines } R_x)$, $(R_i \text{ partially refines } R_x)$ or $(R_x \text{ contains } R_i)$. Therefore, there is no impact on architecture. The following is a change impact example for the change ‘Add Requirement’.

Change Impact Example for the Change ‘Add Requirement’ (Add R_x)

We explain one of the change impact rules for the change type ‘Add Requirement’ with the following example from the RPM requirements document.

Requirement 5 *The system shall store patient blood pressure measured by the sensor in the central storage.*

Requirement 15 *The system shall store patient Central Venous Pressure (CV Pressure) measured by the sensor in the central storage.*

where (Requirement 15 refines Requirement 5)

The stakeholders’ need the following change: Measuring and storing blood pressure is refined further for Pulmonary Artery Pressure (PA pressure). Therefore, we propose the change ‘Add Requirement’ in which the new requirement refines Requirement 5.

Requirement X *The system shall store patient Pulmonary Artery Pressure (PA pressure) measured by the sensor in the central storage.*

where (Requirement X refines Requirement 5)

Figure 7.2 shows the *Satisfies* trace for Requirement 5 and the candidate impacted architectural elements for Requirement X.

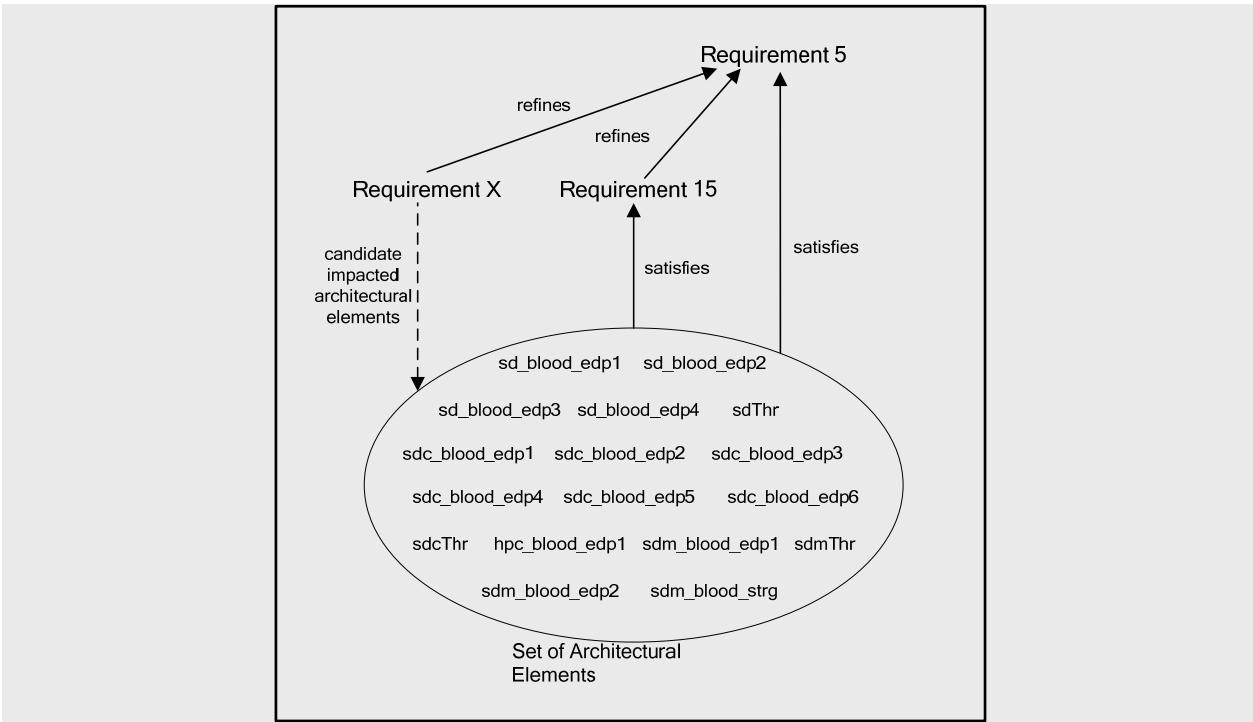


Figure 7.2 Candidate Impacted Architectural Elements for the Added Requirement

Since Requirement X is the refinement of the system property given in Requirement 5, the architectural elements that implement measuring and storing patient blood pressure are candidate impacted for measuring and storing patient PA pressure. Figure 7.3 shows the part of the RPM architecture that satisfies Requirement 5. The architectural elements in Figure 7.3 are the candidate impacted elements given in Figure 7.2.

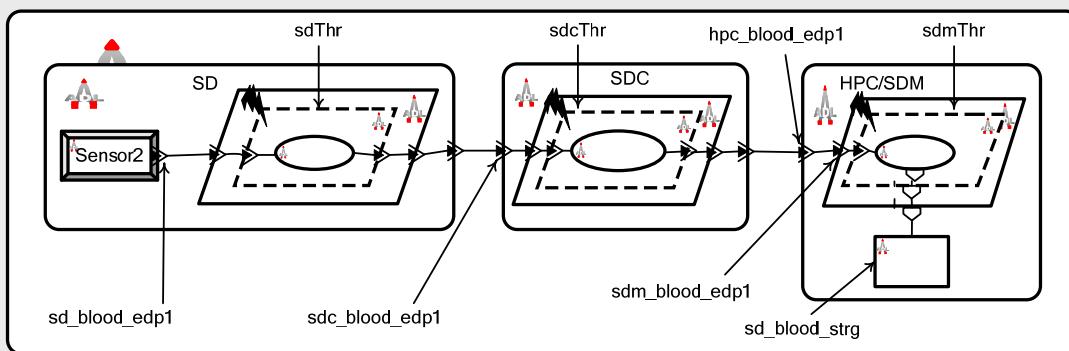


Figure 7.3 Part of the RPM Architecture for Storing Blood Pressure

Before adding Requirement X, Requirement 15 is the only requirement that refines Requirement 5. Therefore, the part of the RPM architecture in Figure 7.3 satisfies the property in Requirement 15 (*Storing patient CV pressure measured by the sensor*). We inspected the architecture based on the new requirement and candidate impacted architectural elements.

We changed the architecture to get the new requirement satisfied by the architecture. Figure 7.4 gives the changed part of the RPM architecture.

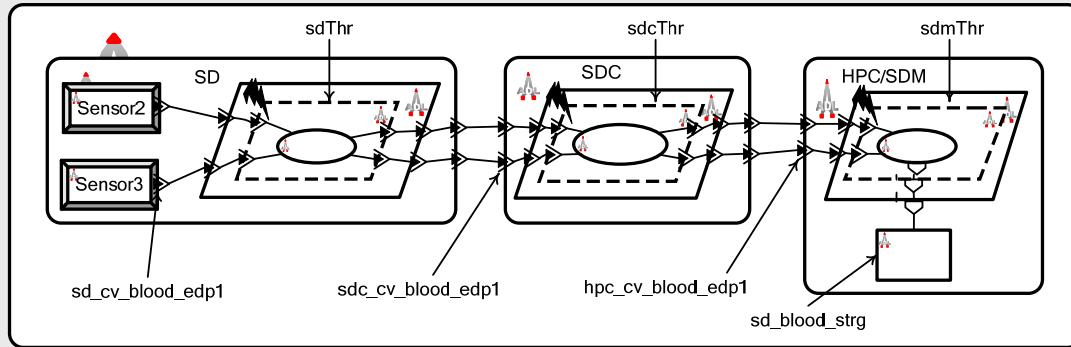


Figure 7.4 Changed Part of the RPM Architecture for Storing Blood Pressure

We added a new sensor (*Sensor 3*) and new event data ports (*sd_cv_blood_edp1*, *sdc_cv_blood_edp1*, and etc.) to measure and transmit the patient CV pressure. The threads *sdThr*, *sdcThr* and *sdmThr* have some of the new event data ports. The measured CV pressure is stored to the existing data store (*sd_blood_strg*). Therefore, according to our changes the actual impacted architectural elements are the threads *sdThr*, *sdcThr*, *sdmThr* and the data store *sd_blood_strg*.

To implement the change, the software architect takes certain design decisions. Furthermore, after performing the change, architecture verification and trace establishment techniques in Chapter 6 have to be applied to verify the new architecture and to generate new traces.

Please note that the candidate impacted architectural elements might not be actually impacted at all. For instance, we could propose new event data ports, new threads, new sensors and new data storages for the example. None of the candidate impacted elements would be affected. With candidate impacted elements we aim at identifying architectural elements that satisfy changed properties and/or existing properties related to the added property.

The following is the explanation of the change impact rule in Table 7.1 for the change ‘Add Requirement’ ($\text{Add } R_x$) where (R_x refines R_i).

Change Impact Rule for ‘Add Requirement’ ($\text{Add } R_x$) where (R_x refines R_i)

Candidate impacted architectural elements for the change type ‘Add Requirement’ ($\text{Add } R_x$)

where (R_x refines R_i)

= Architectural elements traced from R_i are candidate impacted

Explanation:

Let R_i, R_x be requirements and E_A be the set of architectural elements that *satisfies* R_i where P_i and P_x are formulas for R_i and R_x , and P_A is the formula for the system property E_A is needed to implement.

= {By using formalization of the refines relation}

$$P_x \rightarrow P_i$$

= {By using formalization of the satisfies trace}

The fulfillment of P_A implies the fulfillment of P_i

P_i also holds for the set of architectural elements E_A . The new requirement R_x is a refinement of R_i . Usually, the architectural elements in E_A provide part of the functionality that satisfies P_x . The elements in E_A can be reused or adapted in order to implement the new requirement R_x . Therefore, they are candidate impacted architectural elements.

The following is the derivation of the change impact rule in Table 7.1 where the change ‘Add Requirement’ is not a domain change (Add R_x where R_i contains R_x).

Change Impact Rule for ‘Add Requirement’ (Add R_x) where (R_i contains R_x)

Candidate impacted architectural elements for the change type ‘Add Requirement’ (Add R_x)

where (R_i contains R_x)

= No candidate impacted architectural element

Derivation:

Let RM be a requirements model where P_{RM} is the formula for RM .

The requirements model RM is the set of requirements R_1, R_2, \dots, R_k where P_1, P_2, \dots, P_k are formulas for R_1, R_2, \dots, R_k , and $k \geq 1$. P_{RM} can also be represented in the following way:

$$P_{RM} = P_1 \wedge P_2 \wedge \dots \wedge P_k$$

Please note that if the requirements R_1, R_2, \dots, R_k are written as formulas $\forall x\varphi_1, \forall x\varphi_2, \dots, \forall x\varphi_k$ with $\varphi_1, \varphi_2, \dots, \varphi_k$ in CNF, we have the following: ($P_{RM} = \forall x(\varphi_1 \wedge \varphi_2 \wedge \varphi_3 \wedge \dots \wedge \varphi_k)$).

Let R_i and R_x be requirements where P_i and P_x are formulas for R_i and R_x , and ($i \leq k$)

Let RM' be the requirements model after the change ‘Add R_x ’ where $P_{RM'}$ is the formula for RM' .

= {By using formalization of the change type ‘Add Requirement’}

$$P_{RM'} = P_{RM} \wedge P_x$$

If P_{RM} and P_x are written as formulas $\forall x(\varphi_1 \wedge \varphi_2 \wedge \varphi_3 \wedge \dots \wedge \varphi_k)$ and $\forall x\varphi_x$ with $\varphi_1, \varphi_2, \dots, \varphi_k, \varphi_x$ in CNF, we have the following: ($P_{RM}^l = \forall x(\varphi_1 \wedge \varphi_2 \wedge \varphi_3 \wedge \dots \wedge \varphi_k \wedge \varphi_x)$).

= {By using formalization of the contains relation}

We have the following: ($P_i = P_x \wedge P^l$) where P^l denotes properties that are not captured in P_x . Please note that if the requirements R_i and R_x are written as formulas $\forall x\varphi_i$ and $\forall x\varphi_x$ with φ_i and φ_x in CNF and P^l is expressed as $\forall x\psi$ with ψ in CNF, we understand the following: R_i contains R_x iff ($P_i = \forall x(\varphi_x \wedge \psi)$), and ($\neg(\forall x(\varphi_x \rightarrow \varphi_i))$) and ($\neg(\forall x(\psi \rightarrow \varphi_i))$) are satisfiable.

$$P_{RM}^l = P_{RM} \wedge P_x$$

$$P_{RM}^l = \forall x(\varphi_1 \wedge \varphi_2 \wedge \varphi_3 \wedge \dots \wedge \varphi_k \wedge \varphi_x)$$

$$P_{RM}^l = \forall x(\varphi_1 \wedge \varphi_2 \wedge \dots \wedge \varphi_i \wedge \dots \wedge \varphi_k \wedge \varphi_x)$$

$$P_{RM}^l = \forall x(\varphi_1 \wedge \varphi_2 \wedge \dots \wedge \varphi_x \wedge \psi \wedge \dots \wedge \varphi_k \wedge \varphi_x)$$

$$P_{RM}^l = \forall x(\varphi_1 \wedge \varphi_2 \wedge \dots \wedge \varphi_x \wedge \psi \wedge \dots \wedge \varphi_k)$$

$$P_{RM}^l = \forall x(\varphi_1 \wedge \varphi_2 \wedge \dots \wedge \varphi_i \wedge \dots \wedge \varphi_k)$$

$$P_{RM}^l = \forall x(\varphi_1 \wedge \varphi_2 \wedge \varphi_3 \wedge \dots \wedge \varphi_k)$$

Then we get $P_{RM}^l = P_{RM} = \forall x(\varphi_1 \wedge \varphi_2 \wedge \varphi_3 \wedge \dots \wedge \varphi_k)$

= {By using the formalization of domain changes and refactoring}

Properties that are captured in the requirements model RM are preserved in the new requirements model RM^l and there is no new property in the new requirements model RM^l . Therefore, we can conclude that the architecture, that satisfies requirements in the requirements model RM , satisfies requirements in the requirements model RM^l after the change ‘Add R_x ’. There is no need to change the architecture and therefore, there is no candidate impacted architectural element.

7.3.2 Candidate Impacts for Other Changes

The changes ‘Delete Requirement’ and ‘Update Requirement’ (except ‘Add Property to Requirement’) update existing properties described in requirements. Architectural elements that satisfy the changed properties are candidate impacted. The changed requirements may have properties not changed. Architectural elements that satisfy the unchanged properties are not impacted. The propagation path is traversed in order to identify the impacted requirement(s) which has no unimpacted properties (if possible) or at least which has the

smallest number of unimpacted properties. Architectural elements that are traced from the identified requirement(s) are named candidate impacted.

We define a function for traversing the propagation path. The function *traversePropagationPath* takes a change type, a requirement to which the change is introduced, a set of relations of the requirement used in the propagation path and a set of all traces between R&A as input. The output of the *traversePropagationPath* function is a set of architectural elements which are candidate impacted by the requirements change. The following is the algorithm of the function.

```

1  traversePropagationPath(ChangeType c, Requirement r, Set srlp, Set st): Set {
2      ChangeType pc = empty
3      Set srl = empty-set
4      Set rlp = empty-set
5
6      If (srlp is empty-set) {
7          sae = getArchitecturalElements(r, st)
8          Return sae
9      }
10
11     If (c is ‘Delete Requirement’) {
12         Boolean i = false
13         ForEach relation rl ∈ srlp {
14             If ((rl is ‘refines’) AND (rl.target is r)) {
15                 i = true
16                 pc = getPropagatedChange(c, r, rl)
17                 srl = getRelations(rl.source)
18                 rlp = getRelationsInPropagation(pc, rl.source, srl)
19                 sae = sae + traversePropagationPath(pc, rl.source, rlp, st)
20             }
21         } // End of ForEach
22
23         If (i = false) { sae = getArchitecturalElements(r, st) }
24         Return sae
25     }

```

```

26
27     Boolean k = false
28     ForEach relation rl ∈ srlp {
29         If ((rl is ‘refines’) AND (rl.target is r)) OR
30             ((rl is ‘partially refines’) AND (rl.target is r)) {
31                 k = true
32                 pc = getPropagatedChange(c, r, rl)
33                 srl = getRelations(rl.source)
34                 rlp = getRelationsInPropagation(pc, rl.source, srl)
35                 sae = sae + traversePropagationPath(pc, rl.source, rlp, st)
36             } else {
37                 If ((rl is ‘contains’) AND (rl.source is r)) {
38                     k = true
39                     pc = getPropagatedChange(c, r, rl)
40                     srl = getRelations(rl.target)
41                     rlp = getRelationsInPropagation(pc, rl.target, srl)
42                     sae = sae + traversePropagationPath(pc, rl.target, rlp, st)
43                 }
44             }
45     } // End of ForEach
46
47     If (k = false) { sae = getArchitecturalElements(r, st) }
48
49     Return sae
50 } // End of traversePropagationPath function
51

```

The function is recursive. It returns the set of candidate impacted architectural elements when there is no relation to be traversed in the propagation path (see lines 6 - 9). If there is any relation in the propagation path, the function checks the type of the change and relation to identify candidate impacts based on the change impact rules in Table 7.2. Table 7.2 has change types in the rows and relation types in the columns. Please note that the relation in the column is considered for the change in the row only if the requirements relation is in the propagation path. If the change is ‘Delete Requirement’ and the deleted requirement is refined by another requirement in the propagation path, the function continues to traverse

the path for the refining requirement (see lines 11 - 21). If there is no refining requirement in the propagation path, architectural elements that satisfy the deleted requirement are identified as candidate impacted (line 23) (see the row *Delete R_i* in Table 7.2).

The function checks if the updated requirement is (partially) refined by another requirement or contains another requirement in the propagation path. If there is any refining/contained requirement in the propagation path, the function continues to traverse the path for the refining/contained requirement (see lines 28 - 45). If there is no refining/contained requirement in the propagation path, architectural elements that satisfy the updated requirement are identified as candidate impacted (line 47).

Table 7.2 Traversal Rules for Change Types “Delete Requirement” and “Update Requirement”

Changes	Requirements Relation Types					
	R _i contains R _k	R _i refines R _k	R _i partially refines R _k	R _k contains R _i	R _k refines R _i	R _k partially refines R _i
Delete R _i	Do not traverse the propagation path for R _k	Do not traverse the propagation path for R _k	Do not traverse the propagation path for R _k	Do not traverse the propagation path for R _k	Take R _k to traverse the propagation path	Do not traverse the propagation path for R _k
$-pt$ R _i \mapsto R' _i	Take R _k to traverse the propagation path	Do not traverse the propagation path for R _k	Do not traverse the propagation path for R _k	Do not traverse the propagation path for R _k	Take R _k to traverse the propagation path	Take R _k to traverse the propagation path
$pt \rightarrow pt^l$ R _i \mapsto R' _i	Take R _k to traverse the propagation path	Do not traverse the propagation path for R _k	Do not traverse the propagation path for R _k	Do not traverse the propagation path for R _k	Take R _k to traverse the propagation path	Take R _k to traverse the propagation path
$+ct$ R _i \mapsto R' _i	Take R _k to traverse the propagation path	Do not traverse the propagation path for R _k	Do not traverse the propagation path for R _k	Do not traverse the propagation path for R _k	Take R _k to traverse the propagation path	Take R _k to traverse the propagation path
$-ct$ R _i \mapsto R' _i	Take R _k to traverse the propagation path	Do not traverse the propagation path for R _k	Do not traverse the propagation path for R _k	Do not traverse the propagation path for R _k	Take R _k to traverse the propagation path	Take R _k to traverse the propagation path
$ct \rightarrow ct^l$ R _i \mapsto R' _i	Take R _k to traverse the propagation path	Do not traverse the propagation path for R _k	Do not traverse the propagation path for R _k	Do not traverse the propagation path for R _k	Take R _k to traverse the propagation path	Take R _k to traverse the propagation path

In Table 5.2 in Chapter 5 we give the change impact alternatives for the change propagation. Each cell in Table 5.2 gives change alternatives in order to propagate the changes in the rows by using the relations in the columns. The requirements engineer selects one of the

alternatives to propagate the change. The changes in the rows of Table 7.2 represent the changes selected by the requirements engineer.

The following is an example for change impact for the change ‘Add Constraint to Property of Requirement’.

Change Impact Example for the Change ‘Add Constraint to Property of Requirement’

We explain one of the change impact rules for the change type ‘Add Constraint to Property of Requirement’ with the following requirements from the RPM requirements document.

Requirement 4 *The system shall store patient temperature measured by the sensor in the central storage.*

Requirement 7 *The system shall warn the doctor when the temperature threshold is violated.*

Requirement 8 *The system shall generate an alarm if the temperature threshold is violated.*

Requirement 9 *The system shall show the doctor the temperature alarm at the doctors’ computers.*

Requirement 14 *The system shall store patient temperature measured by the sensor in the central storage and it shall warn the doctor when the temperature threshold is violated.*

Figure 7.5 shows the part of the RPM requirements model for the requirements above.

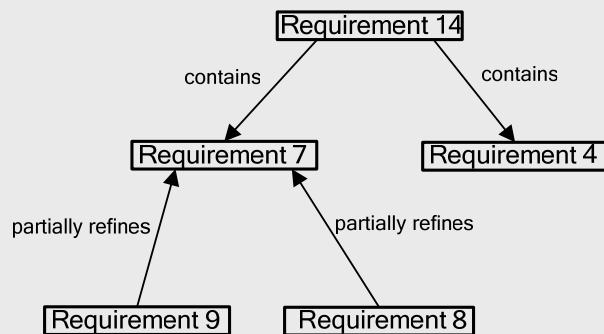


Figure 7.5 Part of the RPM Requirements Model

The stakeholders’ need a change for the RPM system: The system shall warn the doctor with the information about the patient’s condition when the temperature threshold is violated. The change ‘Add Constraint to Property of Requirement’ is proposed for Requirement 14.

Proposed Change: Add Constraint to Property of Requirement 14

Description of the Proposed Change: If the temperature threshold is violated, the system shall warn the doctor with the information about the patient’s condition.

The property of Requirement 14 is ‘warning the doctor about the temperature threshold violation’. The constraint added to the property of Requirement 14 is ‘warning the doctor with the information about the patient’s condition’. The proposed change is propagated to the requirements which contain or refine the property ‘warning the doctor about the temperature threshold violation’ (see Chapter 5). Figure 7.6 shows the propagation path of the proposed change for Requirement 14.

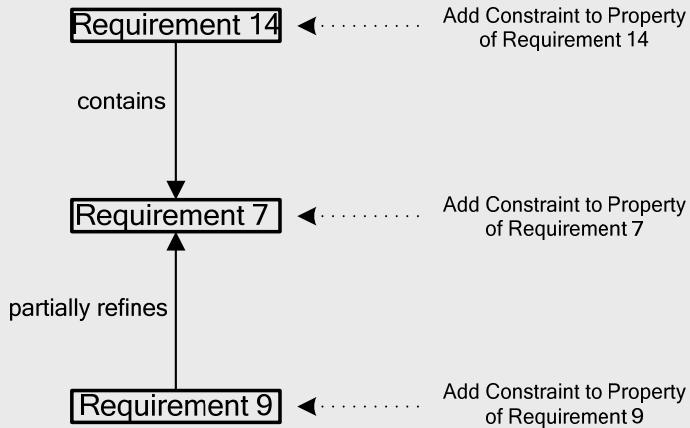


Figure 7.6 Propagation Path of the Proposed Change for Requirement 14

The proposed changes for Requirement 7 and Requirement 9 in the propagation path are the following:

Proposed Change for Requirement 7: Add Constraint to Property of Requirement 7

Description of the Proposed Change: If the temperature threshold is violated, the system shall warn the doctor with the information about the patient’s condition.

Since the property of Requirement 14, which has the proposed change, is contained by Requirement 7, the same proposed change is introduced to Requirement 7.

Proposed Change for Requirement 9: Add Constraint to Property of Requirement 9

Description of the Proposed Change: The system shall show the doctor the temperature alarm with information about the patient’s condition at the doctor’s computer.

Since the property of Requirement 7, which has the propagated change, is partially refined by Requirement 9, the same proposed change is introduced to Requirement 9.

The proposed changes in the propagation path are the ‘Add Constraint to Property of Requirement’ change. Therefore, in order to identify the candidate impacted architectural elements for the proposed change in Requirement 14, we traverse the propagation path in Figure 7.6 based on the rules in Table 7.2.

According to Table 7.2, R_k is taken to traverse the propagation path if $(R_i \mapsto R_i^{+ct})$ and $(R_i \text{ contains } R_k)$. Since Requirement 14 has the change ‘Add Constraint to Property of Requirement’ and Requirement 14 *contains* Requirement 7, Requirement 7 is taken to traverse the propagation path.

According to Table 7.2, R_k is taken to traverse the propagation path if $(R_i \mapsto R_i^{+ct})$ and $(R_k \text{ partially refines } R_i)$. Since Requirement 9 has the change ‘Add Constraint to Property of Requirement’ and Requirement 9 *partially refines* Requirement 7, Requirement 9 is taken to traverse the propagation path. There is no other requirement which has the proposed change in the propagation path. Therefore, architectural elements traced from Requirement 9 are candidate impacted. Figure 7.7 shows the propagation path and candidate impacted architectural elements.

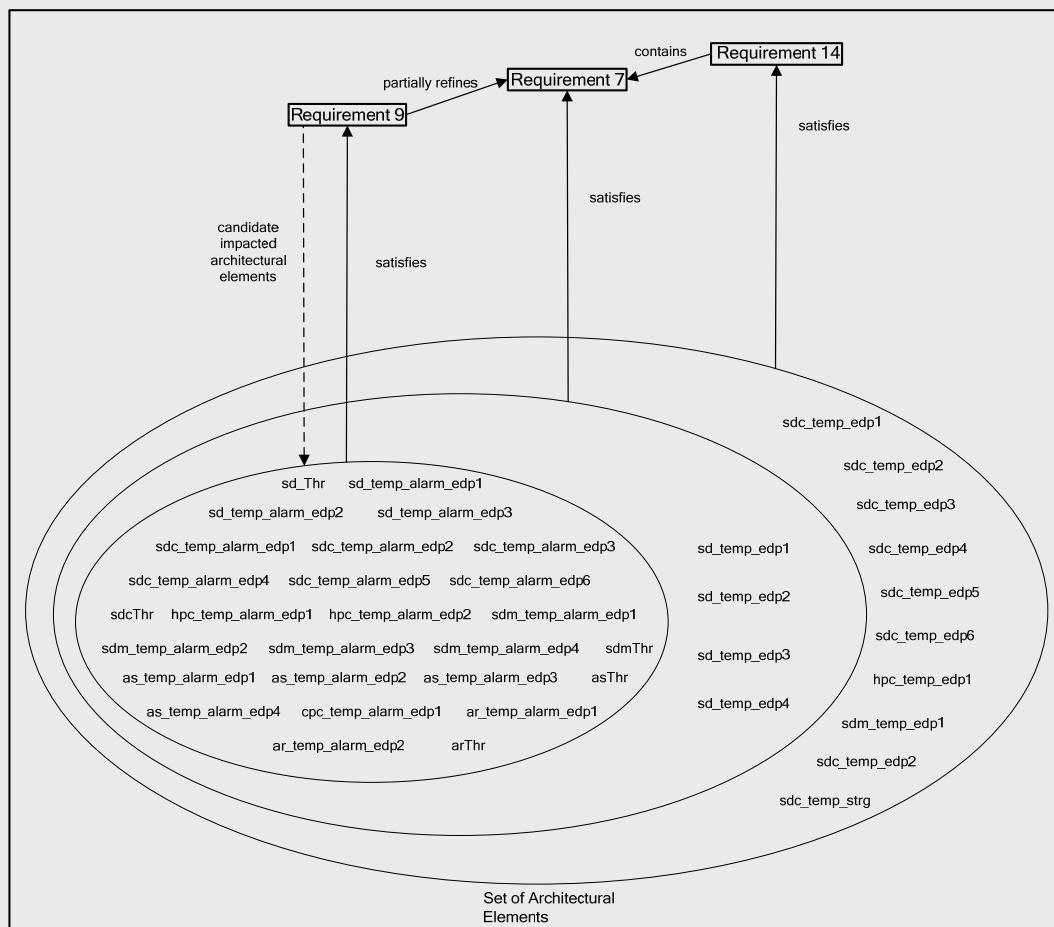


Figure 7.7 Candidate Impacted Architectural Elements for the Constraint Added to Requirement 14

Requirement 9 has the most refined property related to the proposed change. Therefore, in order to implement the change proposed for Requirement 14, architectural elements that satisfy Requirement 9 are identified as candidate impacted.

Traversal rules in Table 7.2 are derived from the semantics of change types and requirements relations. The following is the derivation of the traversal rule in Table 7.2 for the change ‘Add Constraint to Property of Requirement’ for R_i ($R_i \xrightarrow{+ct} R_i^l$) where (R_k partially refines R_i).

Traversal Rule for the Change ‘Add Constraint to Property of Requirement’

Candidate impacted architectural elements for the change type ‘Add Constraint to

Property of Requirement’ for R_i ($R_i \xrightarrow{+ct} R_i^l$)

where (R_k partially refines R_i) and the change is propagated to R_k ($R_k \xrightarrow{+ct} R_k^l$)
= Take R_k to traverse the propagation path

Derivation:

Let R_i be a requirement where P_i is the formula for R_i . P_i is represented in a conjunctive normal form (CNF) in the following way:

$$P_i = \forall x (p_1 \dots p_n); n \geq 1 \text{ and } p_i \text{ is disjunction of literals}$$

Let R_k be a requirement where P_k is the formula for R_k .

Let R_i^l and R_k^l be the requirements after the changes ($R_i \xrightarrow{+ct} R_i^l$) and ($R_k \xrightarrow{+ct} R_k^l$) where P_i^l and P_k^l are the formulas for R_i^l and R_k^l .

Let E_{Ai} be the set of architectural elements that *satisfies* R_i and E_{Ak} be the set of architectural elements that *satisfies* R_k where P_{Ai} is the formula for the system property E_{Ai} is needed to implement and P_{Ak} is the formula for the system property E_{Ak} is needed to implement.

= {*By using formalization of the satisfies trace*}

The fulfillment of P_{Ai} implies the fulfillment of P_i

The fulfillment of P_{Ak} implies the fulfillment of P_k

= {*By using formalization of the partially refines relation*}

$P_k = \forall x (p_1^l \dots p_z^l); z < n \text{ and } \forall x (p_j^l \rightarrow p_j) \text{ for all } j \in 1..z$

= {*By using formalization of the change type ‘Add Constraint to Property of Requirement’ for R_i* }

$P_i^l = \forall x ((p_1^l \dots p_t^l) \wedge (p_{t+1}^l \dots p_n^l)); t \leq z \text{ and } \forall x (p_j^l \rightarrow p_j) \text{ for all } j \in 1..t$

The properties captured in $\forall x(p_{z+1} \dots p_n)$ in R_i are not affected by the change. These properties are not captured by R_k . Therefore, the propagation path is traversed for R_k .

The software architect identifies the candidate impacted architectural elements with tool support. Then, he/she starts investigating the impacted architectural elements and changing the software architecture for the changed requirements. In the following we explain how to identify possible changes for software architecture when the software architecture does not satisfy the changed requirements.

7.4 Proposing Architectural Changes

With the first part of our technique, the software architect identifies software architecture elements that are candidate impacted by the requirements changes. He/she analyzes and possibly changes parts of the architecture. After the changes, the software architecture may not satisfy the changed requirements. The second part of our technique is to propose possible changes in the software architecture when the software architecture does not satisfy the changed requirements. The technique is based on architecture verification. The output of the verification is a counter example if the requirements are not satisfied. The counter example is used together with a classification of architectural changes to propose changes in the software architecture. These changes produce a new version of the architecture that possibly satisfies the changed requirements.

We provide a change impact function for proposing architectural changes. It takes a changed requirement, a set of traces between the requirement and software architecture, and a counter example where the requirement is not satisfied, as input. The function produces a set of proposed architectural changes as output. The following is the signature of the change impact function.

$$\text{impact} : SR \times SST \times SCE \rightarrow SSAC$$

where SR is the set of requirements, SST is the set of sets of traces, SCE is the set of counter examples and $SSAC$ is the set of sets of architectural changes.

Counter example is an ordered set of states which are generated when the requirement is not satisfied. There are no transition rules applicable in the last state of the counter example. A state transition rule is fired if its left-hand side pattern matches in the current state. The next state is formed based on the right-hand side of the transition rule. The idea is to make such changes in the architecture that will make the application of some transition rules possible.

Application of changes may happen iteratively until the requirement is eventually satisfied. In the analysis of the counter example we have the following limitations and assumptions.

- *Analyzing the counter example in our approach is limited to the operational semantics of AADL in [197] [198].* This semantics mostly deals with passing & storing data in a data flow, dispatching & executing threads and switching modes. These specific details are used when proposed architectural changes are identified. Therefore, architectural changes in our approach may not be generalized for other architecture description languages and other versions of semantics.
- *Architectural changes in our approach are limited to the possible missing parts of the architecture for mainly data flow and thread execution.* Designing architecture based on requirements is a creative process. There are an infinite number of designs that satisfy the requirements for a given project. Therefore, the number of changes over the architecture is infinite. We do not consider changes such as adding new systems, processes or threads which may cause infinite number of solutions for the changed requirements.
- *It is assumed that there is a next state from the last state of the counter example.* It is possible that the last state might be the final state where no state transition is fired further. In this case, the software architect should check all the states in the counter example to change the architecture. Even if the last state is not the final state, changing the architecture to enable a next state may not produce an architecture that satisfies the changed requirement. The software architect may need iterations of changing and verifying the architecture.

The following is an illustration of the idea:

Example for Proposing Architectural Changes

The example was already used in Section 7.3.1. It shows a change request to illustrate how we identify the candidate impacted architectural elements for the change ‘Add Requirement’.

Requirement X *The system shall store patient Pulmonary Artery Pressure (PA pressure) measured by the sensor in the central storage.*

Figure 7.8 gives the changed part of the RPM architecture that implements the added requirement.

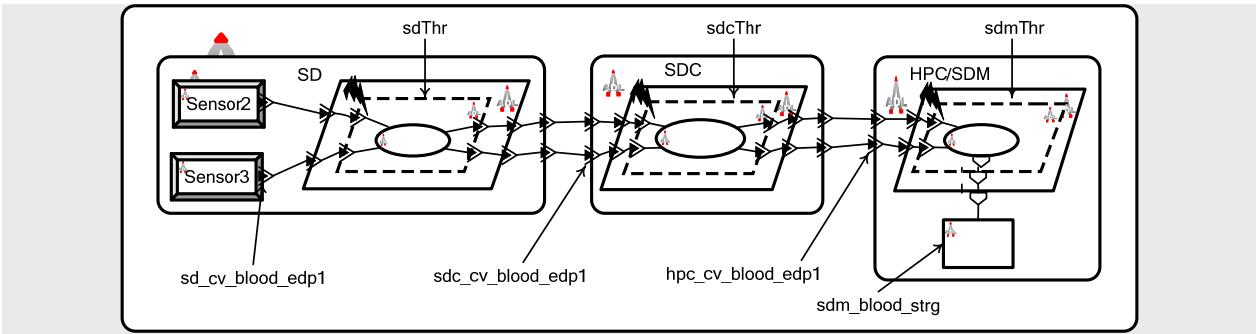


Figure 7.8 Changed Part of the RPM Architecture for Storing Blood Pressure

We added a new sensor (*Sensor 3*) and new event data ports (`sd_cv_blood_edp1`, `sdc_cv_blood_edp1`, `hpc_cv_blood_edp1` and etc.) to measure and transmit the patient PA pressure. The threads `sdThr`, `sdcThr` and `sdmThr` have some of the new event data ports. The measured PA pressure is stored to the existing data store (`sd_blood_strg`).

Sensor 3 measures and transmits the patient PA pressure via the event data ports (`sd_cv_blood_edp1`, `sdc_cv_blood_edp1`, `hpc_cv_blood_edp1` and etc.) and the threads (`sdThr`, `sdcThr` and `sdmThr`). The following is the LTL formula for the added requirement (see Chapter 6 for the reformulation of requirements as LTL formulas):

LTL formula in Maude: `(mc initializeThreads({ MAIN system Wholesys . imp }) |=u <> ((MAIN -> hpc -> sdm -> sdmTh) @ bloodStored) .)`

The formula states that if the *DI* data instance is contained by the data port `sd_cv_blood_edp1` of *Sensor 3*, then eventually in the future the state in the state transition system in the `sdmTh` thread is set to the `bloodStored` state (the *DI* data instance is stored by the `sdm_blood_strg` data store of the *SDM* component). Please note that the *DI* data instance is created in the initial state by a test thread in the RPM model. Therefore, the LTL formula does not explicitly indicate the *DI* data instance and the `sd_cv_blood_edp1` data port of *Sensor 3*. The formula creates the initial state instead. After executing the model checker in Maude, the LTL formula is false and it returns the counter example.

During the design of the architecture, the software architect assigns some ‘AllocatedTo’ traces between the requirement and parts of the software architecture that are supposed to satisfy the requirement. Some ‘AllocatedTo’ traces are generated between the new requirement and parts of the architecture that are used in the verification (see Chapter 6).

There might be unexpected architectural elements used in the verification. We compare the assigned and generated ‘AllocatedTo’ traces before we analyze the last state of the counter example. If there is any unexpected element in the generated traces, we have to find

out why it is used in the verification. The unexpected elements might be the cause that the requirement is not satisfied. If some elements in the assigned traces are not in the generated traces, some elements have not been used in the verification yet. If the application of further rules is possible after the change, the rest of the elements in the assigned traces may be used. Figure 7.9 shows the assigned and generated ‘AllocatedTo’ traces for the added requirement.

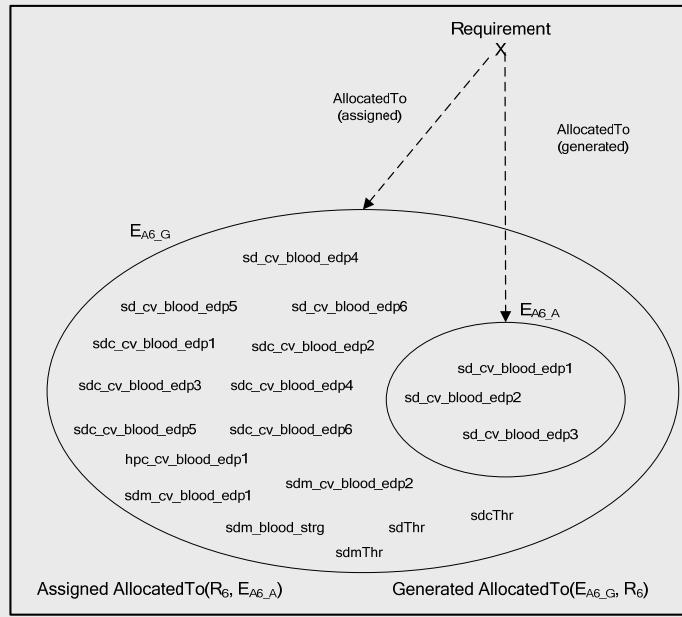


Figure 7.9 Assigned and Generated ‘AllocatedTo’ Traces for the Added Requirement

The set of architectural elements used in the verification (the generated ‘AllocatedTo’) is a subset of the architectural elements to which Requirement X is allocated (the assigned ‘AllocatedTo’). Therefore, there is no unexpected architectural element used in the verification. Figure 7.10 gives the last state of the counter example.

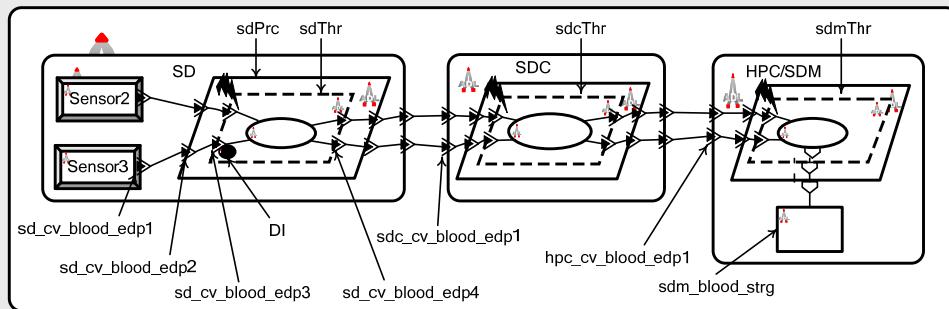


Figure 7.10 Last State of the Counter Example in the First Check

In Figure 7.10, the *DI* data instance is at the buffer of the *sd_cv_blood_edp3* data-in-port of the *sdThr* thread. Before the last state of the counter example, the data is passed to the

sd_cv_blood_edp3 data-in-port of the *sdThr* thread from the *sd_cv_blood_edp2* data-in-port of the *sdPrc* process. The transmission of messages from source port to destination port along connections is modeled as *equations* in Maude [197] [198]. The following equation models the transmission of a data along a level-down connection $C.P \rightarrow C.C1.P1$ from the *P* data-in-port of the *C* component to the *P1* data-in-port of the *C1* subcomponent.

```

1   op transfer : MsgList -> MsgList [ctor] .
2
3   vars C C1 : ComponentId .           vars P P1 : PortId .
4   vars PORTS PORTS2 OTHER-COMPONENTS : Configuration .
5   vars ML ML' : MsgList .           var CONXS : ConnectionSet .
6
7   eq < C : Component |
8       features :
9       < P : InPort | buffer : transfer(ML) > PORTS,
10      subcomponents :
11      < C1 : Component | features : < P1 : InPort | buffer : ML' > PORTS2 >
12      OTHER-COMPONENTS,
13      connections : (P --> C1 . P1) ; CONXS >
14      =
15      < C : Component |
16      features : < P : InPort | buffer : nil > PORTS,
17      subcomponents :
18      < C1 : Component | features : < P1 : InPort | buffer : ML' :: transfer(ML) >
19      PORTS2 >
20      OTHER-COMPONENTS .

```

As a result of applying the equation in our example, the *sd_cv_blood_edp3* data-in-port has the *DI* data instance (line 18), and the *sd_cv_blood_edp2* data-in-port's buffer is empty (line 16).

According to the AADL standard, the only possible transition is *thread dispatching* if the data is at the buffer of (event) data-in-port of the thread. The *sdThr* thread is an *aperiodic* thread. Therefore, the state transition rule for aperiodic thread dispatching can be fired. An architectural change has to make the application of the state transition rule for aperiodic thread dispatching possible. Aperiodic thread dispatching is modeled by the following conditional rewrite rule in Maude [197] [198]:

```

1  var O : ThreadId .           var P : PortId.      var PROGRAM : ThreadBehaviour .
2  var MTS : ModeTransitionSystem .  var TN : ThreadName .  var IMPL : ImpleName .
3  var PORTS : Configuration .     vars ML ML' : MsgList .
4
5  crl [aperiodic-incoming-message] :
6    < O : Thread | properties : aperiodic-dispatch ; TP,
7      used : U,
8      modes : MTS,
9      deactivated : false,
10     features :
11       (< P : InEventDataThreadPort | buffer : ML :: transfer(ML') >
12         PORTS),
13     status : completed,
14     behavior : PROGRAM,
15     threadType : TN, implementationType : IMPL >
16   =>
17   < O : Thread | used : true,
18     features :
19     dispatchInputPorts(
20       < P : InEventDataThreadPort | buffer : ML :: ML' > PORTS),
21     status : active >
22   if someTransEnabled(transitions(TN, IMPL), PROGRAM,
23     dispatchInputPorts(
24       < P : InEventDataThreadPort | buffer : ML :: ML' >
25         PORTS)) .

```

The left-hand side pattern of the transition rule is in lines 6 - 15 and lines 22 - 26. To fire the *aperiodic-incoming-message* transition rule, the following conditions should hold:

- (1) the thread is *active* (line 9),
- (2) the thread status is in *complete* (line 13)
- (3) some of the transitions in the behavioral annex of the thread are enabled (lines 22-23)
- (4) there is an incoming data at the buffer of the data-in-port of the thread (lines 23 - 24)

There is already a data at the the buffer of the *sd_cv_blood_edp3* data-in-port of the *sdThr* thread. Therefore, architectural changes should be proposed to make the conditions 1, 2 and

3 hold. In order to make the conditions 1, 2 and 3 hold, there are two changes on the architecture: *changing the mode of the thread* and *changing the behaviour of the thread*. The thread might be activated and its status might be set to *completed* by changing the mode of the thread. The behaviour of the thread is coded as states and state transitions with its activation and status in the behavioral annex. Please note that the states and state transitions in the behavioral annex are different than the states and state transitions in the model checker. Changing the behaviour of the thread (the behavioral annex of the thread) may make the thread *active* and its status *complete*. If none of the transitions in the thread is enabled (see the condition 3), either some of the transitions in the behavioral annex or the mode of the thread is changed. The thread may have different states and transitions in different modes.

Let's inspect the requirement, the software architecture and the possible changes. The conditions 1 and 2 hold for the *sdThr* thread. The thread is *active* and its status is in *complete*. None of the transitions in the behavioral annex of the thread is enabled because the states and transitions in the *sdThr* thread are about the data received from Sensor 1 (not shown in Figure 7.10) and Sensor 2. There is no state and transition handling the data which is received from Sensor 3 and put to the the *sd_cv_blood_edp3* data-in-port. The *sdThr* thread has no mode. Therefore, we decide to change the behavior of the thread by introducing new states and state transitions in the behavioral annex. We add the following state transition with the new state *cvBloodPassed* to the behavioural annex of the *sdThr* thread:

```
idle -[sd_cv_blood_edp3?(inMessage)]-> cvBloodPassed { sd_cv_blood_edp4!(inMessage); };
```

The added state transition states the following: If the *sdThr* thread is in the *idle* state and receives the measurement data at the *sd_cv_blood_edp3* event data port, then the received data is passed to the *sd_cv_blood_edp4* event data port. We re-execute the model checker over the changed architecture. The LTL formula is again false and it returns another counter example. After the first check of the architecture we have three more iterations that we do not illustrate here because the architectural changes are again changing the behaviour of the thread. We add new states and state transitions to the behavioral annex of the threads *sdThr* and *sdmThr* after the second and third iterations.

As a fourth check we re-execute the model checker over the changed architecture. The LTL formula is true and it returns the execution. Figure 7.11 gives the last state of the execution trace.

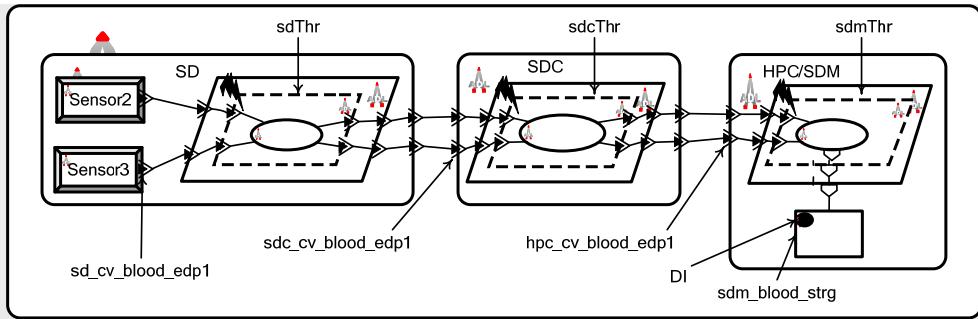


Figure 7.11 Last State of the Execution Trace

In the last state of the execution trace, the *DI* data instance is stored and the *bloodStored* state is reached. Therefore, the architecture satisfies the new requirement. For the changed part of the RPM architecture in Figure 7.8 we have four iterations to make the architecture satisfy the new requirement.

The software architect may always take different architectural decisions to change the architecture. If we use the RPM architecture in Figure 7.12 instead of the one in Figure 7.8, we change the architecture only once.

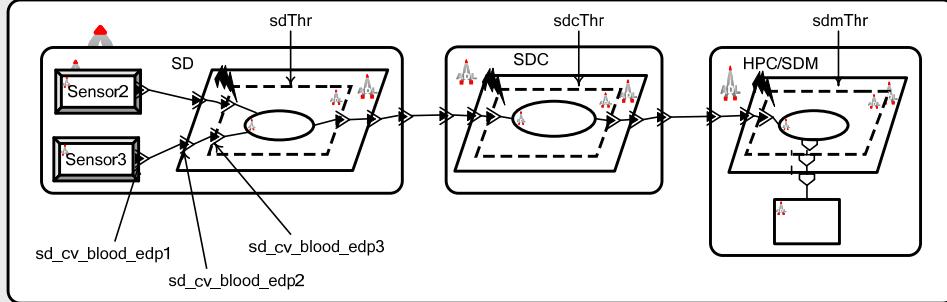


Figure 7.12 Another RPM Architecture for Storing CV Pressure

In Figure 7.12, only Sensor 3 and three data-in-ports (*sd_cv_blood_edp1*, *sd_cv_blood_edp2* and *sd_cv_blood_edp3*) are added to the RPM architecture. Existing data-in-ports and connections are used to transmit the data from the *sdThr* thread to the *sdmThr* thread. The only change over the architecture is changing the behaviour of the *sdThr* thread. Since the *sdThr* and *sdmThr* threads use the existing data-in-ports for the data measured by Sensor 3, they do not need any change in their behavioral annex.

The main steps in the change impact function are the following:

Comparing the Assigned and Generated Traces. There might be unexpected architectural elements used in the verification. We compare the assigned and generated ‘AllocatedTo’ traces before we analyze the last state of the counter example. Figure 7.13 gives the comparison of generated and assigned ‘AllocatedTo’ traces for a requirement.

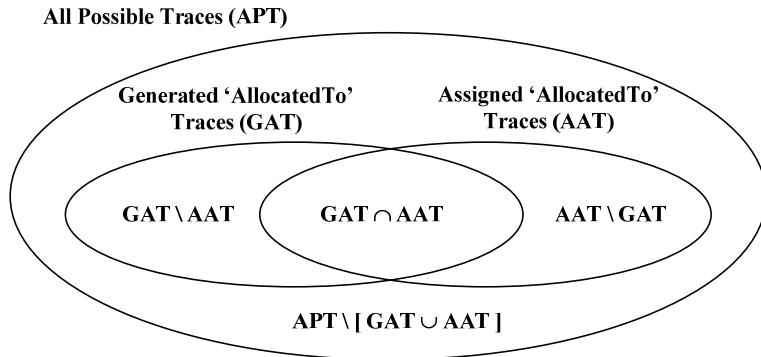


Figure 7.13 Venn Diagram for Generated and Assigned ‘AllocatedTo’ Traces for a Requirement

Figure 7.13 is used to compare the generated and assigned ‘AllocatedTo’ traces:

- If $(GAT \setminus AAT) = \emptyset$, then all architectural elements used in the verification for the requirement are designed to satisfy the requirement. If some elements in the assigned traces are not in the generated traces, some elements have not been used in the verification yet. If the application of further rules is possible after the change, the rest of the elements in the assigned traces may be used. The change impact algorithm takes the second step to analyze the counter example.
- If $(GAT \setminus AAT) \neq \emptyset$, then there are some unexpected elements in the generated traces. We have to find out why they are used in the verification. The unexpected elements might be the cause that the requirement is not satisfied. Therefore, the change impact algorithm does not take the second step.

Analyzing the Counter Example. There are no transition rules applicable in the last state of the counter example. The idea is to make such changes in the architecture that will make the application of some transition rules possible. Our analysis of the counter example is limited to enabling passing data, dispatching threads, executing threads and switching modes. These are the main actions in the dynamic semantics of AADL used in our approach. We have the following steps for analyzing the counter example:

- *Locating the architectural elements that may need changes.* We want to locate the elements that may cause transitions. These are data, data-in-ports, data-out-ports, data storage, threads, systems and processes. The statements in the right-hand side patterns give information where data can be found in the architecture, if a thread is dispatched/executed and if a mode is switched. Table 7.3 gives the categories of the state transition rules in AADL with the right-hand-side patterns.

Table 7.3 Categories of the State Transition Rules in AADL with the Right-hand Side Patterns

Categories of State Transition Rules in AADL	Right-hand Side Patterns
Passing Message M_1	Event/Data M_1 at the buffer of the (event) data-in-port of System S_1
	Event/Data M_1 at the buffer of the (event) data-in-port of Process P_1
	Event/Data M_1 at the buffer of the (event) data-in-port of Thread T_1
	Event/Data M_1 at the buffer of the (event) data-out-port of Device D_1
	Event/Data M_1 at the buffer of the (event) data-out-port of System S_1
	Event/Data M_1 at the buffer of the (event) data-out-port of Process P_1
Dispatching Thread T_1	Event/Data M_1 at the <i>internalbuffer</i> of the (event) data-in-port of Thread T_1 & Thread T_1 is in <i>active</i> status
	Thread T_1 is in the <i>active</i> status
Executing Thread T_1	Event/Data M_1 at the buffer of the (event) data-out-port of Thread T_1 & Thread T_1 is in the <i>completed</i> status
	Thread T_1 is in the <i>completed</i> status
Switching the Mode of Thread T_1	Thread T_1 is in the <i>inactive</i> status
	Thread T_1 is in the <i>completed</i> status

In the first column of Table 7.3, there are four categories of the state transition rules (*Passing Message M_1* , *Dispatching Thread T_1* , *Executing Thread T_1* and *Switching the Mode of Thread T_1*).

- *Matching the last state of the counter example for the right-hand side patterns of the state transition rules.* By analyzing the right-hand side patterns, we know what are the possible locations of the data and the status of threads for thread dispatching and execution. We check the last state of the counter example to find the location of data and activated threads for dispatching and execution in the last state.
- *Analyzing the left-hand side of the state transition rules to propose architectural changes.* Data and dispatched/executed threads in the last state of the counter example are the potential architectural elements to trigger further state transition rules. For instance, if a thread is already dispatched, the next state transition rule is for thread execution. We analyze the left-hand side of the state transition rules to identify the conditions for each rule to be applied. Architectural changes are proposed to make the conditions hold.

In this section we illustrate proposing architectural changes for the counter example where the thread is dispatched. Analysis of all the state transition rules is given in Appendix J. There are two right-hand side patterns for *Dispatching Thread T_1* : (i) if T_1 is *aperiodic*, its status is *active* and event/data M_1 is at the *internalbuffer* of its (event) data-in-port, (ii) if T_1 is *periodic*, its status is *active*. Table 7.4 gives proposed architectural changes that would trigger state

transition rules if the thread is dispatched. The first column of Table 7.4 lists the right-hand side patterns of the state transition rules for *Dispatching Thread T1*. The second column of the table gives the state transition rules which can be fired when a thread is already dispatched. According to the AADL standard, only the state transition rules for *Executing Thread T1* can be fired if the thread is already dispatched. The third column gives the possible architectural changes to make the conditions of the left-hand side patterns of the state transition rules for *Executing Thread T1*.

Table 7.4 Right-hand Side Patterns of the State Transition Rules for Dispatching Thread T1 with Proposed Architectural Changes in AADL

	Right-hand Side Patterns of the State Transition Rules for Dispatching Thread T1	State Transition Rules to be Fired Further	Proposed Architectural Changes
1	Event/Data <i>M1</i> at the <i>internalbuffer</i> of the (event) data-in-port of Thread <i>T1</i> & Thread <i>T1</i> is in the <i>active</i> status	Executing Thread <i>T1</i>	Change the mode of Thread <i>T1</i>
			Change the behaviour of Thread <i>T1</i>
2	Thread <i>T1</i> is in the <i>active</i> status	Executing Thread <i>T1</i>	Change the mode of Thread <i>T1</i>
			Change the behaviour of Thread <i>T1</i>

In row (1), the right hand side pattern is for dispatching an aperiodic thread. In row (2), the right-hand side pattern is for dispatching a periodic thread. To fire the transition rules for both periodic and aperiodic thread execution, the following condition in the left-hand side patterns of the state transition rules should hold: *some of the transitions in the behavioral annex of the thread are enabled*.

If none of the transitions in the behavioral annex of the thread is enabled, either some of the transitions in the behavioral annex or the mode of the thread is changed. The thread may have different states and transitions in different modes. Therefore, there are two changes on the architecture (*changing the mode of the thread* and *changing the behaviour of the thread*) to make the condition hold.

There might be multiple applicable state transition rules which affect different parts of the architecture. Therefore, multiple changes may be proposed for different parts of the

architecture. The software architect should analyze each proposed change and decide which one to implement. We tried the approach with relatively simple state transition rules. We have not studied the applicability of the approach for more complex state transition rules in AADL.

Iterating. Calling the change impact function is iterative: the software architect may continue changing the architecture. The software architect selects one or more of the proposed changes to be implemented. After implementing the proposed architectural changes, the architecture is verified again. If the changed requirement is not satisfied by the changed architecture, the change impact function is called again.

7.5 Tool Support

In Chapter 4 and Chapter 5, we showed the details of our tool named Tool for Requirements Inferencing and Consistency Checking (TRIC). We extended TRIC with features for identifying candidate impacted architectural elements [235]. In this section, we give the details of the tool. Tool support for architecture verification as part of changing software architecture is already given in Chapter 6. In Section 7.5.1, we depict the usage of the tool in the context of a requirements modeling process. Section 7.5.2 describes the main features of the tool with some screenshots.

7.5.1 The Modeling Process

We depict the usage of the tool in a requirements modeling and architecture design process with identifying candidate impacted architectural elements and proposing architectural changes. Figure 7.14 gives a UML activity diagram of the process. Change impact analysis techniques for requirements and software architecture are interleaved. Therefore, the process in Figure 7.14 also contains activities for change impact analysis in requirements models. The activities for change impact analysis in software architecture are *identifying candidate impacted architectural elements, proposing architectural changes and changing architecture model*.

The process consists of the following activities.

Modeling Requirements & Designing Architecture: This activity takes the requirements document as input and produces the Requirements Model (RM), Architecture Model (AM) and Trace Model (TM). The requirements engineer models the requirements in the requirements document by assigning relations between them with tool support in Chapter 4. The software architect designs the software architecture for the requirements and traces between requirements and architecture are assigned and/or generated (see Chapter 6).

Analyzing Change Impact in Requirements Model: The activity takes the change request and the Requirements Model (RM) as input and produces impacted requirements and proposed changes in the requirements model as output. The change is interpreted by the requirements engineer in order to propose and propagate changes in the Requirements Model (RM) (see Chapter 5).

Identifying Candidate Impacted Architectural Elements: The activity takes proposed changes, impacted requirements, Requirements Model (RM), Architecture Model (AM) and Trace Model (TM) as input and produces impacted architectural elements as output. The activity is automatic. The software architect/requirements engineer selects the proposed change in the requirements model. TRIC gives the impacted requirement to be traced to candidate impacted architectural elements for the proposed change. Eclipse model editor is used to display the candidate impacted architectural elements with the Trace Model (TM).

After identifying the candidate impacted architectural elements, the software architect decides to implement the changes in requirements model and architecture model.

Changing Requirements Model: This activity takes the Requirements Model (RM) with proposed changes and impacted requirements as input and produces the New Requirements Model (RM') as output. The activity is manual. The requirements engineer changes requirements according to proposed changes.

Changing Architecture Model: The activity takes the New Requirements Model (RM'), Trace Model (TM), Architecture Model (AM), candidate impacted architectural elements and impacted requirements as input and produces the New Architecture Model (AM') and the New Trace Model (TM') as output. It is a manual activity. The software architect changes the Architecture Model (AM) based on candidate impacted architectural elements in order to make the architecture satisfy new/changed requirements. He also updates the traces (the New Trace Model – TM') between the New Requirements Model (RM') and the New Architecture Model (AM').

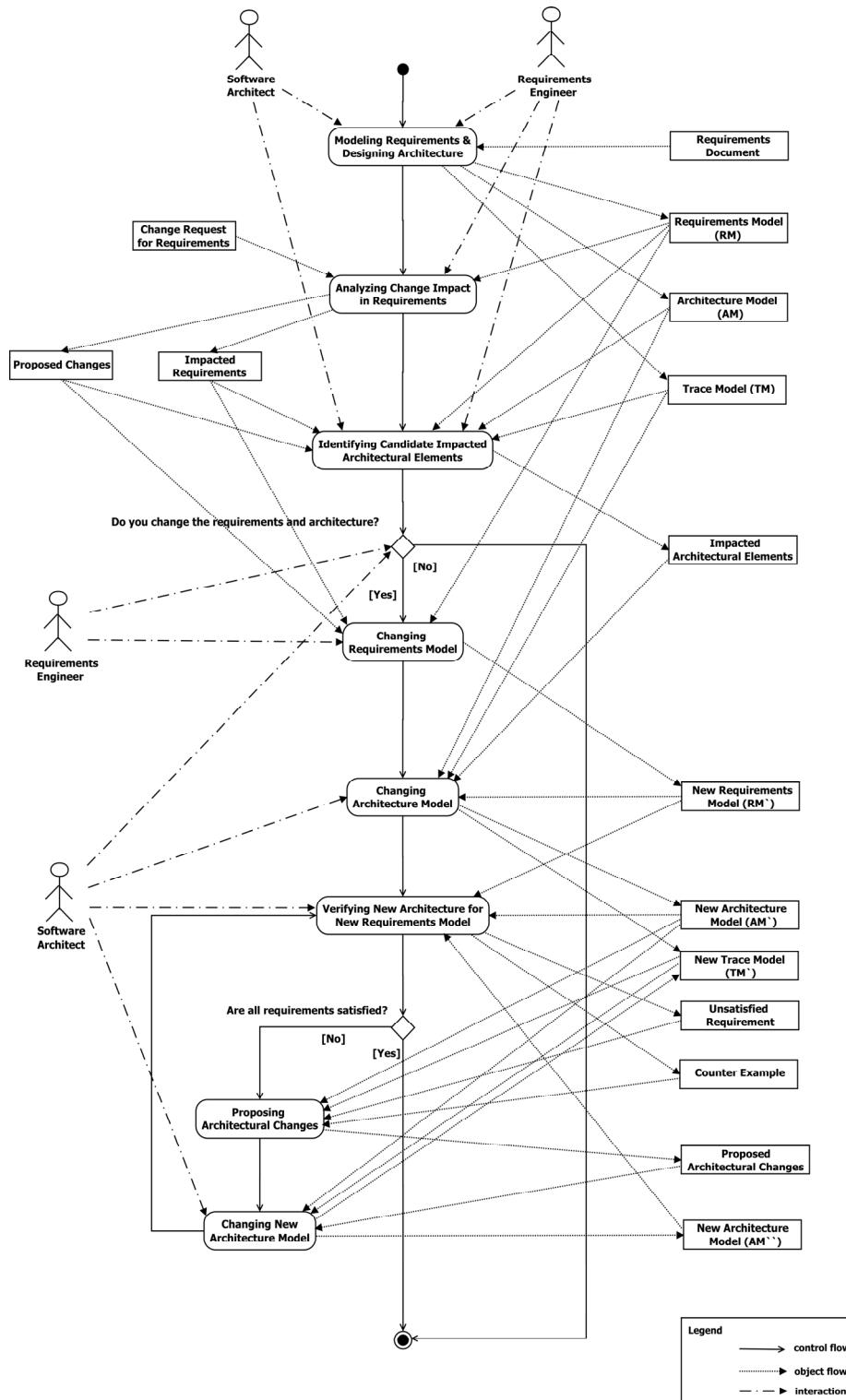


Figure 7.14 Requirements Modeling and Architectural Design Process with Change Impact Analysis

Verifying New Architecture for New Requirements Model: This activity takes the New Requirements Model (RM') and the New Architecture Model (AM') as input and gives an execution trace or a counter example and unsatisfied requirements (if there is any) as output. The activity checks whether the requirements are satisfied by the architecture. It is semi-automatic. The software architect reformulates the new/changed requirements in terms of logical formulas over the architecture. These logical formulas are checked for the architecture by the model checker in Maude automatically.

Proposing Architectural Changes: The activity takes the New Architecture Model (AM'), New Trace Model (TM'), unsatisfied requirement and counter example as input and produces proposed architectural changes as output. The proposed changes are derived by analyzing the latest configuration of the architectural elements in the counter example.

Changing New Architecture Model: The activity takes the New Architecture Model (AM'), New Trace Model (TM'), unsatisfied requirement and proposed architectural changes as input and produces the New Architecture Model (AM'') and the New Trace Model (TM') as output. It is a manual activity. The software architect changes the New Architecture Model (AM') based on proposed architectural changes in order to make the architecture satisfy new/changed requirements. He also updates the traces (the New Trace Model – TM') between the New Requirements Model (RM') and the New Architecture Model (AM'').

Iterating: The process in Figure 7.14 is iterative: the software architect may return to the verification of new architecture for new requirements model activity if the requirements are still not satisfied by the new architecture. If all requirements are satisfied, the process is terminated.

7.5.2 Tool Features

The tool support is a combination of the usage of *TRIC*, *Eclipse Model Editor* and *Maude*. We describe the features of the tool support: *identifying candidate impacted architectural elements* and *proposing architectural changes*.

Identifying Candidate Impacted Architectural Elements: Figure 7.15 gives the GUI for selecting the proposed requirements change which supports the *identifying candidate impacted architectural elements* activity.

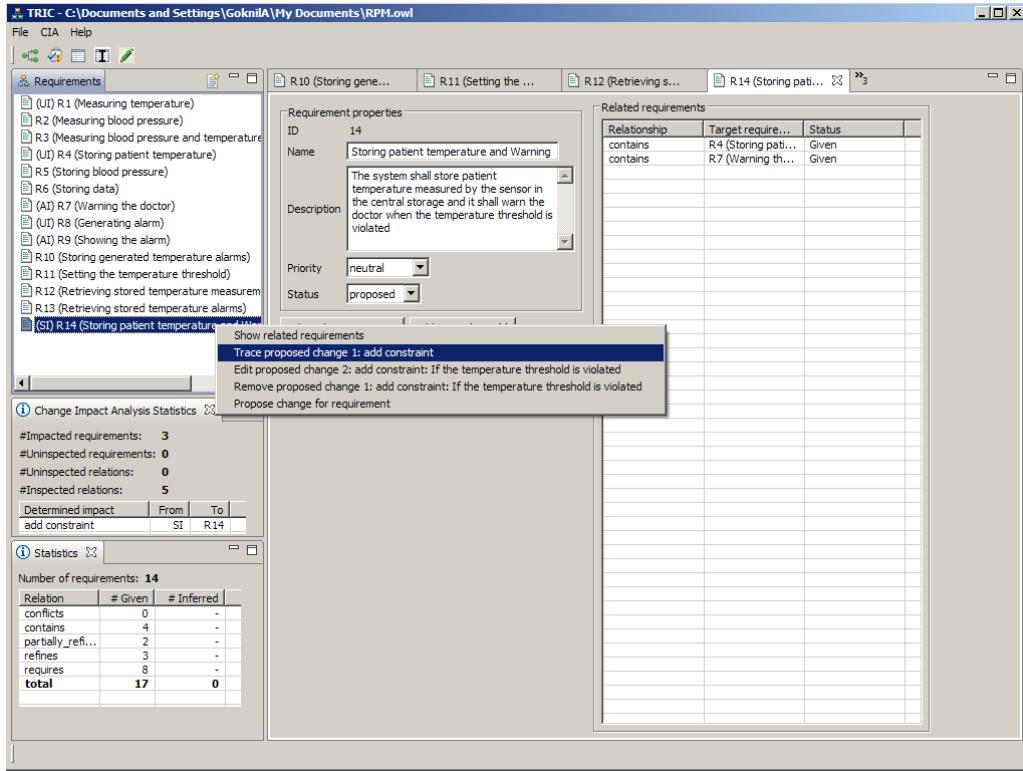


Figure 7.15 GUI for Selecting the Proposed Requirements Change

The left-hand side of the window lists the requirements in the model. The requirements are tagged as *SI – Starting Impacted* and *UI - Unimpacted*. The right-hand side of the window shows the details of the selected requirement (R14). The pop-up menu opened by right clicking on the selected requirement (R14) is used to select the proposed requirements change. After selecting the proposed requirements change, the propagation path for the selected change is traversed by the tool to identify the impacted requirement to be traced to architecture. Figure 7.16 gives the output of traversing the propagation path of the proposed requirements change.

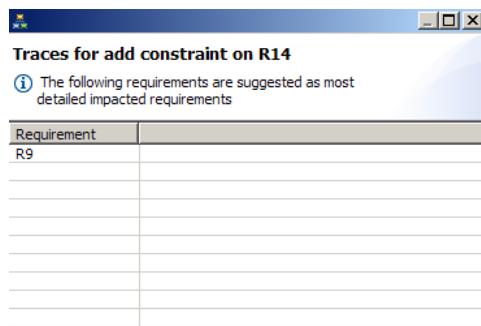


Figure 7.16 Output of Traversing the Propagation Path of the Proposed Requirements Change

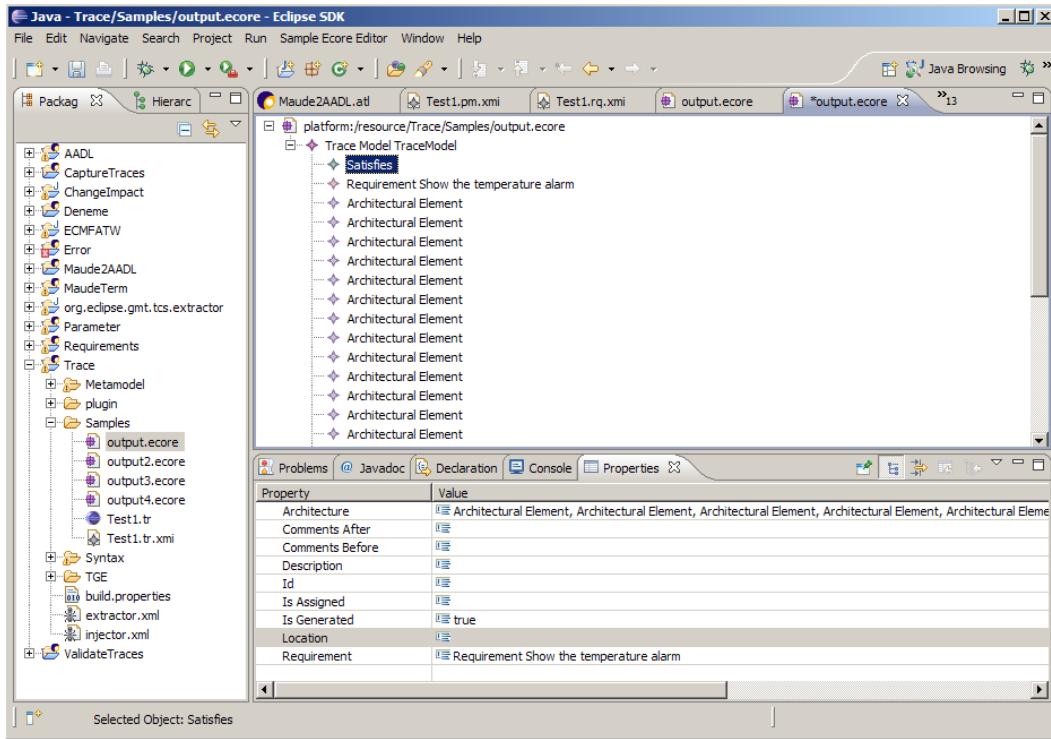


Figure 7.17 Output of the Identifying Candidate Impacted Architectural Elements Activity

Architectural elements traced from the impacted requirement(s) in Figure 7.16 are candidate impacted architectural elements for the requirements change selected in Figure 7.15. The Eclipse Model Editor is used to trace from the impacted requirement to candidate impacted architectural elements by using the trace model (see Figure 7.17).

The right-hand side of the window shows the file *output.ecore* which is the trace model. The trace model includes the traces, requirements and architectural elements that are associated with the traces. The software architect can identify the architectural elements traced from the impacted requirement by using the trace model in the Eclipse Model Editor. The details of the chosen element in the trace model can be seen in the bottom of the window.

Proposing Architectural Changes: Architectural changes are proposed based on counter example which is the output of verification when the reformulated requirements fail. We use the Open-Source AADL Tool Environment (OSATE) – Topcased [204] which includes an AADL front-end and architecture verification capabilities as plug-ins. The plug-in [182] developed by Artur Boronat is used to generate Maude representation of AADL models which can be simulated and verified. In Maude, we verify the software architecture for reformulated requirements in LTL. We use Eclipse plug-in developed in the context of MOMENT2 [30] to run Maude under Windows. We do not have a tool support to analyze

the counter example and to propose changes yet. Analysis of the counter example is currently done manually according to the algorithm in Section 7.4.

7.6 Related Work

We discuss related work in two categories: *Change Impact Analysis in Software Architectures* and *Tool Support*.

7.6.1 Change Impact Analysis in Software Architectures

A number of approaches in the literature address change impact analysis in software architectures. Jonsson and Lindvall [133] present common strategies for performing change impact analysis. They divide the strategies into two categories: *automatable* (traceability/dependency analysis and slicing techniques) and *manual* (design documentation and interviews). Automatable impact analysis strategies often employ algorithmic methods for change propagation [133]. Traceability and dependency analyses differ in scope and detail level. Traceability analysis is the analysis of the relations among all types of artifacts, while dependency analysis focus on relations extracted from the source code. Since our approach analyzes requirements relations and traces between R&A, it can be considered as traceability analysis.

Algorithmic analysis is employed by Lee et al. [153] in order to compute the impact of changes on object-oriented software. Lee et al. uses data dependency graphs with a classification of changes for object-oriented software to determine the impacted elements in object-oriented source code. The approach addresses the impact analysis in source code, not in high-level design. Briand et al. [35] [36] presents a change impact analysis approach for UML analysis/design models. The changes between two versions of UML models are automatically identified based on a change classification. Then, model elements impacted by the changes are identified by using formally defined change impact analysis rules expressed in OCL. Similar to our approach, the approach in [35] [36] provides resulting changes for the impacted model elements. On the other hand, change impact analysis rules in [35] [36] are specific to UML models and changes in requirements are not considered. Tang et al. [241] introduce Architecture Rationale and Element Linkage (AREL) model represented as a Bayesian Belief Network (BBN). AREL captures the causal relationship between architectural elements and decisions using probabilities. This allows architects to perform change impact in software architecture based on probability theory. The input probabilities have to be entered by the software architect based on previous experience. The main difference with our approach is that the approach in [241] provides only impacted architectural elements without any proposed change. Han [106] introduces an approach for

impact analysis and change propagation based on dependencies of software artifacts. Propagation rules are defined based on change patterns. A change pattern includes initial modifications, consequent modifications with Boolean expressions that state the dependencies of the elements involved. Han applies the approach in order to determine the consequent modifications in design and source code for the initial modifications in design. Our approach supports determining impacted architectural elements with consequent changes for changes in requirements.

Westhuizen and Hoek [248] provides an approach for propagating architectural changes within a product line architecture. The approach has two algorithms. The first one is a differencing algorithm that automatically calculates the difference between two versions of a product line architecture. The second algorithm is a merging algorithm that propagates the changes captured by the differencing algorithm to the second product line architecture. The merging algorithm requires the presence of some common elements among the architectures. It propagates the changes from one architecture to another. Our approach focuses on the propagation of changes in requirements to software architecture.

Slicing techniques are mainly developed to understand dependencies using independent slices of the program [85]. Slicing is based on data and control flows in the program. Slicing techniques limit change propagation by identifying the scopes of changes. The work by Tip et al. [244] is an example of slicing techniques for C++ programs. Architectural slicing introduced by Zhao et al. [265] [266] is similar to program slicing. As opposed to program slicing, architectural slicing runs on the software architecture. The approach determines one slice of the software architecture for the proposed change. Components that might be impacted by the changed component are traced by using a graph of information flows. Therefore, the approach requires all the information flows of the software architecture being exposed. The main difference between the architectural slicing and our approach is that our approach identifies candidate impacted architectural elements with possible architectural changes caused by changes in requirements. Zhao et al. mainly focus on the questions such as 'If a change is made to a component c, what other components might be affected by c?'. Feng and Maletic [80] address the propagation of architectural changes within the same architecture. Their approach can be considered as both dependency analysis and slicing technique. Interface and method slicing are used together with analysis of component dependencies.

7.6.2 Tool Support

Some requirements management tools support change impact analysis in software architectures. The selection of tools is based on INCOSE management tool survey [124].

IBM Rational RequisitePro [119] provides a matrix view to show the traces between requirements and architectural elements. When a requirement is changed, traces of the changed requirement are marked as suspect. All architectural elements directly or indirectly related to the changed requirement are candidate impacted. The software architect has to inspect the candidate impacted architectural elements to identify changes if there is any.

Borland Caliber [27] supports one trace type between artifacts. It is a trace that can be established between any two artifacts such as requirements model and software architecture. Change impact analysis is manual. Similar to RequisitePro, Borland Caliber provides traceability matrix and traceability diagram to represent traces. All architectural elements directly or indirectly related to the changed requirement are candidate impacted. Therefore, the software architect should inspect all directly and indirectly related architectural elements by using traceability matrix and diagram manually.

TopTeam Analyst [246] identifies suspected traces for change impact analysis. However, direct traces are not automatically marked as suspect when a requirement is changed. All traces for the changed requirement have to be selected and marked as suspect manually to identify the candidate impacted architectural elements. On the other hand, it is possible to get subscribed to specific elements in artifacts. When one of these elements such as a requirement is changed, the subscribers get a message. The message contains the name of the element, the user who changed the element and a link to the element for a quick inspection. IBM Telelogic Doors [120] supports a manual analysis of the relations and requirements affected by a change. When a requirement is changed, its traces are marked as suspect automatically.

All industrial tools given above supports marking traces as suspected for changed requirements. All direct and indirect traces of the changed requirement are marked as suspect. Therefore, in these tools, all architectural elements directly or indirectly related to the changed requirement are candidate impacted. None of the inspected industrial tools provides proposing possible changes for software architecture to make the architecture satisfy the new and/or changed requirements.

7.7 Conclusions

We presented a technique for change impact analysis in software architecture. Our technique has two parts that use the approaches in Chapters 5 and 6. In the first part, we use the formal semantics of requirements relations and traces between R&A to identify the candidate impacted architectural elements. Most of the approaches and tools like IBM

Rational RequisitePro and DOORS do not focus on formal semantics of requirements relations and traces. By using formal semantics, we provide a more precise change impact analysis in software architecture by elimination of false positive impacts. We extended TRIC for identifying candidate impacted architectural elements. The second part of the technique is to propose possible changes for software architecture when the software architecture does not satisfy changed requirements. We provided a classification of architectural changes. The technique is based on architecture verification. The output of verification is a counter example if the requirements are not satisfied. The counter example is used together with the classification of architectural changes in order to propose changes in the software architecture. The technique supports the architect to change the architecture.

There are some certain limitations and assumptions in our technique. Analyzing the counter example in our approach is limited to the operational semantics of AADL in [197] [198]. This semantics mostly deals with passing & storing data in a data flow, dispatching & executing threads and switching modes. Architectural changes in our approach may not be generalized for other architecture description languages and other versions of semantics.

Architectural changes in our approach are limited to the possible missing parts of the architecture for mainly data flow and thread execution. There are an infinite number of designs that satisfy the requirements for a given project. We do not consider changes such as adding new systems, processes or threads which may cause infinite number of solutions for the changed requirements.

It is assumed that there is a next state from the last state of the counter example. It is possible that the last state might be the final state where no state transition is fired further. Even if the last state is not the final state, changing the architecture to enable a next state may not produce an architecture that satisfies the changed requirement. The software architect may need iterations of changing and verifying the architecture.

In this chapter, we answer *Research Question 4 (How to model requirements, software architecture and traces with their semantics for change management?)* and *Research Question 5 (How can be a change in a requirement propagated to other requirements and to software architecture? How can we support the requirements engineer and software architect for performing changes? How can we formally check if the evolved architecture satisfies evolved requirements?)* raised in Chapter 1. The use of semantics of requirements relations and traces between R&A with tool support addresses the propagation of a change from a requirement to architectural elements. The proposed changes derived from the analysis of the counter example help the software architect to perform changes on software architecture.

There are still some open issues. Since we applied the approach to a limited number of requirements in the Remote Patient Monitoring System requirements document, the results may not be generalizable. We still need to apply the approach to a number of industrial case studies and to obtain empirical results. Our tool needs improvement for usability. The core parts of the tool for identification of candidate impacted architectural elements are implemented. However, the integration of these parts (TRIC and the Eclipse model editor) is currently done manually and we need a user interface to control all these parts. Furthermore, we do not have a tool support to analyze the counter example and to propose changes yet. Analysis of the counter example is currently done manually.

Chapter 8

8 Conclusions

This chapter gives the overall conclusions of the thesis. We outline the problems addressed in this thesis, together with our solution and future research directions.

8.1 Introduction

This chapter gives the overall conclusions of the thesis. First, in Section 8.2 we summarize the problems addressed in the thesis: (i) *explosion of impacts in requirements for requirements changes*, (ii) *manual, expensive and error prone trace establishment between requirements and architecture*, and (iii) *explosion of impacts in software architecture for requirements changes*. We reflect on the solutions for these problems in Section 8.3. Section 8.4 gives further research directions.

8.2 Problems

In this thesis, we have addressed the following problems in change impact analysis for requirements and software architecture:

- **Explosion of Impacts in Requirements for Requirements Changes.** When a change is introduced to a requirement, there might be other requirements impacted by the introduced change. The requirements engineer traces impacted requirements from the changed requirement by using relations among requirements. In practice, requirements documents are often textual artifacts with implicit structure. Most of the relations among requirements are not given explicitly. There is a lack of precise definition of relations among requirements in most tools and approaches. By using only the structural information of relations, the requirements engineer may conclude

that all requirements in the model are impacted. Without considering semantics of relations, change impact analysis may produce high number of false positive and false negative impacts.

- **Manual, Expensive and Error Prone Trace Establishment between Requirements and Architecture.** Once the requirements engineer analyzes the impact of a change in requirements, the software architect needs to identify the impact of this change in software architecture. Traces are needed to be established between Requirements (R) & Architecture (A) in order to identify the impacted parts of the architecture. Designing architecture based on requirements is a problem solving process that relies on human experience and creativity, and is mainly manual. Therefore, trace information may remain implicit and the software architect may need to manually assign traces between R&A. Manual trace establishment is time-consuming, expensive and error prone. The assigned traces might be incomplete and invalid.
- **Explosion of Impacts in Software Architecture for Requirements Changes.** In most approaches, there is a lack of precise definition of traces between R&A. By using only the structural information of traces between R&A, the software architect may conclude that all architectural elements in the architecture are impacted. Without considering semantics of traces, change impact analysis may produce high number of false positive and false negative impacts.

8.3 Solutions

In this section, we explain how we have addressed the aforementioned problems. The proposed techniques tackle impacts explosion and trace establishment issues at early stages of software development life cycle (requirements analysis and architecture design).

- **A modeling language for definition of requirements models.** To give an explicit structure to requirements and their relations, we propose a requirements modeling language. The language is defined according to the MDE principles by defining a metamodel. It is based on a survey about the most commonly found requirements types and relation types. With this language, the requirements engineer can explicitly specify the requirements and the relations among them. We assign relation types with formal semantic definitions in First-Order Logic (FOL) in order to enable reasoning about requirements relations. We use the formal definitions for consistency checking of relations and for inferring new relations. The tool TRIC has been built to support

both reasoning activities. The language supports only textual requirements. There is no support for other requirements artifacts like use case and activity diagrams. On the other hand, the requirements metamodel can be customized in order to apply inferencing and consistency checking to current requirements modeling approaches like SysML and goal-oriented requirements engineering. In [96], we presented the customization of the requirements metamodel for SysML.

- **A change impact analysis technique for requirements.** The technique uses the formal semantics of requirements relations and requirements change types. A classification of requirements changes based on the structure of a textual requirement is given and formalized. The semantics of requirements change types is based on FOL. We support three activities for impact analysis. First, the requirements engineer proposes changes according to the change classification before implementing the actual changes. Second, the requirements engineer identifies the propagation of the changes to related requirements. The change alternatives in the propagation are determined based on the semantics of change types and requirements relations. Third, possible contradicting changes are identified. We provide a tool support for these activities. The tool automatically determines the change propagation paths, checks the consistency of the changes, and suggests alternatives for implementing the change. By the use of change alternatives and propagation paths, some false positive impacted requirements are eliminated. We provide a more precise change impact analysis in requirements models than requirements management tools like IBM RequisitePro. The definitions of the requirements relations do not give information about the structure of properties in a requirement. The requirements engineer has to inspect the requirements to know this. Therefore, the technique provides change alternatives in change propagation to be chosen by the requirements engineer. Change alternatives are used only if there is any requirement related to the changed requirement.
- **A technique for trace establishment between R&A.** The technique provides trace establishment by using architecture verification together with semantics of requirements relations and traces. We use a trace metamodel with commonly used trace types. The semantics of traces is formalized in FOL. Software architectures are expressed in the Architecture Analysis and Design Language (AADL). AADL is provided with a formal semantics expressed in Maude. The Maude tool set allows simulation and verification of architectures. The first way to establish traces is to use architecture verification techniques. A given requirement is reformulated as a property in terms of the architecture. The architecture is executed and a state space is

produced. This execution simulates the behavior of the system on the architectural level. The property derived from the requirement is checked by the Maude model checker. Traces are generated between the requirement and the architectural components used in the verification of the property. The second way to establish traces is to use the requirements relations together with the semantics of traces. Requirements relations are reflected in the connections among the traced architectural elements. Therefore, new traces are inferred from existing traces by using requirements relations. We use semantics of requirements relations and traces to both generate/validate traces and generate/validate requirements relations. The technique is supported by a tool. The tool provides the following: (1) generation/validation of traces by using requirements relations and/or verification of architecture, (2) generation/validation of requirements relations by using traces. We enhance trace establishment between R&A with automation and trace validation. We conducted performance and scalability tests of the tool for generating and validating traces. We focused on model checking part of our tool in the performance and scalability tests. According to the test results, the tool performs well in general. The main limitation of the technique is that it is not possible to explicitly state which property in a complex requirement fails when the requirement has multiple properties. The technique aims at preserving the requirements relations in their implementation in the architecture. There might be some cases where extra dependencies not identified in the requirements analysis are determined in the architecture. In these cases, the software architect should update the requirements model by introducing new relations to the requirements model. We use the formal semantics of behavioral subset of AADL models in Maude implemented by Olveczky et al. [197] [198]. Since AADL standard specification does not define a formal semantics, the semantic definitions in Maude involve an interpretation of what the informal and sometimes ambiguous descriptions in the AADL standard mean. The tool uses AADL and Maude but the technique can be applied with another architecture description language and model checker, provided that the formal semantics of the language is given.

- **A change impact analysis technique for software architecture.** The technique is semi-automatic and requires participation of the software architect. It has two parts. The first part is to identify the architectural elements that implement the system properties to which proposed requirements changes are introduced. By having the formal semantics of requirements relations and traces, we identify which parts of software architecture are impacted by a proposed change in requirements. The

second part of our technique is to propose possible changes for software architecture when the software architecture does not satisfy the new and/or changed requirements. The technique is based on architecture verification. The output of verification is a counter example if the requirements are not satisfied. The counter example is used together with a classification of architectural changes in order to propose changes in the software architecture. These changes produce a new version of the architecture that possibly satisfies the new or the changed requirements. By eliminating some false positive impacts and proposing architectural changes, we provide a more precise change impact analysis in software architecture than requirements management tools like IBM RequisitePro and DOORS.

8.4 Future Research Directions

This thesis explained various applications of semantics of traces to solve the impacts explosion problems. These applications lead to open issues that we will investigate in the future.

- **Change impact analysis for non-functional requirements.** In the change impact analysis technique for requirements we do not consider the distinction between functional and non-functional requirements. There might be different relation types for non-functional requirements like performance and security requirements. Non-functional requirements might be stated in a Domain Specific Language (DSL) rather than in FOL. We plan to select one or two non-functional requirements and investigate their relations within the context of change impact analysis.
- **Extension of requirements metamodel.** The requirements metamodel have the generic entities requirement and requirements relation types. The requirements reasoning technique in Chapter 4 and change impact analysis technique in Chapter 5 do not address specific requirements management approaches like goal-oriented requirements engineering. In order to apply our techniques for requirements management approaches found in the literature, the requirements metamodel needs to be customized. We need an extension mechanism for the requirements metamodel and TRIC. In [96], we presented a possible customization of the requirements metamodel for SysML but we did not study how we can extend TRIC for the customization of the metamodel.
- **Change impact analysis within architectural models.** In the thesis, we propagate the change in a requirement to other requirements and software architecture.

Architectural elements impacted by requirements changes are identified and architectural changes are proposed. However, identification of architectural elements impacted by changes on architecture is not studied in the thesis. We plan to address this issue by applying the formal definitions of architectural elements and their dependencies to change impact analysis. Architectural elements and their dependencies can be formalized in a similar way to requirements and their relations. The formalization can be used to identify architectural elements impacted by a change in an architectural element.

- **Tracing from requirements to architecture, detailed design and source code for change impact analysis.** The change impact analysis techniques should be applied further for other software development artifacts such as detailed design and source code. To identify impacted parts of detailed design and source code, we need tracing from requirements to architecture, detailed design and source code.
- **Reasoning about requirements and architectural design decisions.** Decisions taken in the design of the architecture can be considered as intermediate artifacts between requirements and software architecture. In practice, the focus is mainly on the results of the architectural design (the architectural elements). The alternative decisions and the rationale behind the decisions are easily lost. TRIC can be extended with a metamodel for architectural decisions. Capturing these decisions may facilitate an early assignment of traces between requirements and architecture.
- **Tooling.** We have tool support for the techniques developed within the context of the thesis. The change impact analysis techniques and the requirements modeling language are supported by TRIC. Trace establishment between R&A is based on model transformations in ATL and term-writing logic in Maude. We have two improvements for tooling as a future work. The first one is the improvement of our trace establishment tool support for usability. The core parts of the tool are implemented in ATL and Maude. However, integration of these parts is currently done manually and we need a user interface to control all these parts. The second future improvement is the integration of the tool for trace establishment with TRIC. In the current tooling, we do not have a user interface to control TRIC and the trace establishment tool in a uniform way.

Samenvatting

Softwaresystemen worden steeds complexer. De eisen - *requirements* - waaraan softwaresystemen moeten voldoen veranderen voortdurend en vaak komen er nieuwe eisen bij. Nieuwe en/of aangepaste software-eisen dienen te worden geïntegreerd met de bestaande eisen. Bovendien moeten de software-architectuur en programmacode eveneens worden aangepast. *Change management* is het proces van integratie van veranderde eisen en de aanpassing van het softwaresysteem. De complexiteit van softwaresystemen maakt dit proces kostbaar en tijdrovend. Teneinde de kosten van veranderingen te reduceren is het belangrijk deze veranderingen zo vroeg mogelijk in het softwareontwikkelproces door te voeren.

Traceerbaarheid van software-eisen - *requirements traceability* - is cruciaal voor het in stand houden van de consistentie tussen software-artefacten, dat zijn softwaredocumenten zoals architectuur, ontwerp, code, testen. Traceerbaarheid is de mogelijkheid om software-eisen terug te voeren naar belanghebbenden en deze eisen te koppelen aan corresponderende software-artefacten. Wanneer veranderingen in de eisen worden voorgesteld dan kunnen de gevolgen van deze veranderingen getraceerd worden naar andere software-artefacten zodat kan worden vastgesteld welke delen veranderd moeten worden. *Change impact analysis* is het bepalen van de gevolgen van veranderingen in eisen op andere artefacten. De impact kan betrekking op verschillende software-artefacten. Wij zullen ons in het bijzonder richten op impact van veranderende eisen op de software-architectuur.

De noodzaak van change impact analysis geldt zowel voor software-eisen zelf als voor de software-architectuur. Wanneer een verandering wordt voorgesteld in een software-eis dan dient de *requirement engineer* na te gaan of ook andere software-eisen moeten worden aangepast. Nadat deze impact is vastgesteld dient de software-architect vast te stellen welke elementen in de software-architectuur veranderd moeten worden. Dit is mogelijk door de veranderde software-eisen te traceren naar de software-architectuur. Het handmatig uitvoeren van traceren is moeilijk, duur en foutgevoelig. Er zijn softwarepakketten

ontwikkeld om de change impact analysis te automatiseren (zoals IBM Rational RequisitePro en DOORS). In de meeste van deze pakketten worden relaties - *traces* - tussen software-artefacten vastgelegd, maar de semantiek van deze relaties wordt verder niet uitgewerkt. Hierdoor wordt een veranderde eis al snel gekoppeld – direct of indirect – aan veel mogelijk te veranderen elementen in de architectuur. De requirements engineer dient al deze elementen, die kandidaat zijn om veranderd te worden, ook allemaal te inspecteren en na te gaan of een verandering echt noodzakelijk is.

In dit proefschrift behandelen we een aantal problemen die naar voren komen bij het uitvoeren van de change impact analysis van software-eisen en software-architectuur.

- Het groot aantal software-eisen dat mogelijk beïnvloed wordt door een verandering in een eis (*requirements impact explosion*).
- De *foutgevoeligheid* en *kostbaarheid* van het handmatig bepalen van traces tussen software-artefacten.
- Het groot aantal elementen in de software-architectuur dat mogelijk beïnvloed wordt door een verandering in een eis (*architecture impact explosion*).

We beschrijven een aanpak waarin deze explosies van impacts in software requirements (R) en software-architectuur (A) worden gereduceerd. Deze aanpak is gebaseerd op een welgedefinieerde semantiek van de traces. We gaan ervan uit dat iedere relatie tussen software-artefacts of elementen in deze artefacten een trace kan zijn die gebruikt kan worden in change impact analysis.

De aanpak wordt uitgewerkt in de context van *Model Driven Engineering (MDE)*. MDE behandelt verschillende software-artefacten op een uniforme wijze als modellen. Dit maakt het mogelijk over artefacten te redeneren als modellen. Voor het op deze manier structureren van software-eisen, architectuur en traces worden *metamodellen* gebruikt met een formeel gedefinieerde semantiek.

Dit proefschrift levert de volgende onderzoeksbijdragen:

- Een taal voor het *modelleren* van software-eisen – en hun onderlinge relaties – met een formele semantiek. De consistentie van de modellen kan automatisch worden gecheckt met een daarvoor ontwikkelde software-applicatie (TRIC - *Tool for Requirements Inferencing and Consistency Checking*).

- Een *techniek* voor *change impact analysis* van veranderde software-eisen gebaseerd op relaties tussen de eisen en een classificatie van veranderingen. Deze techniek wordt ondersteund in TRIC.
- Een *techniek* voor *het bepalen van trace relaties* tussen software-eisen (R) en de architectuur (A) gebaseerd op verificatietechnieken voor software-architecturen en de semantiek van de relaties tussen R&A.
- Een *techniek* voor *change impact analysis* van software-architectuur, eveneens gebaseerd op verificatietechnieken voor software-architecturen en de semantiek van de relaties tussen R&A.

REFERENCES

- [1] from <http://www.aptest.com/glossary.html#S>
- [2] Abdoul, T., Champeau, J., Dhaussy, P., Pillain, P. Y., & Roger, J. C. (2008). AADL Execution Semantics Transformation for Formal Verification. *13th International Conference on Engineering of Complex Computer Systems, IEEE Computer Society*, 263-268.
- [3] Abi-Antoun, M., & Aldrich, J. (2008). Static Conformance Checking of Runtime Architectural Structure. *Carnegie Mellon University Technical Report, CMU-ISR-08-132*.
- [4] Abi-Antoun, M., & Medvidovic, N. (1999). Enabling the Refinement of a Software Architecture into a Design. *UML'99, LNCS(1723)*, 17-31.
- [5] Ackermann, C., & Lindvall, M. (2006). Understanding Change Requests to Predict Software Impact. *30th Annual IEEE/NASA Software Engineering Workshop*, 66-75.
- [6] Aizenbud-Reshef, N., Paige, R. F., Rubin, J., Shaham-Gafni, Y., & Kolovos, D. S. (2005). Operational Semantics for Traceability. *ECMDA-TW 2005*, 7-14.
- [7] Aldrich, J., Chambers, C., & Notkin, D. (2002). Architectural Reasoning in ArchJava. *ECOOP'02, LNCS(2374)*, 334-367.
- [8] Allen, R., Douence, R., & Garlan, D. (1998). Specifying and Analyzing Dynamic Software Architectures. *FASE'98, LNCS(1382)*, 21-37.
- [9] Allen, R., & Garlan, D. (1997). A Formal Basis for Architectural Connection. *ACM Trans. Softw. Eng. Methodol.*, 6(3), 213-249.
- [10] Almeida, J. P. A., Dijkman, R., Pires, L. F., Quartel, D., & van Sinderen, M. (2006). Model-Driven Design, Refinement and Transformation of Abstract Interactions. *Int. Jour. of Coop. Inf. Sys.*, 15, 599-632.
- [11] Almeida, J. P. A., Iacop, M. E., & van Eck, P. (2007). Requirements Traceability in Model-Driven Development: Applying Model and Transformation Conformance. *Inf. Syst. Front.*, 9, 327-342.
- [12] Amelunxen, C., Königs, A., Rötschke, T., & Schürr, A. (2008). Metamodeling with MOFLON. *Applications of Graph Transformations with Industrial Relevance LNCS(5088/2008)*, 573-574.
- [13] Antoniol, G., Canfora, G., Casazza, G., De Lucia, A., & Merlo, E. (2002). Recovering Traceability Links between Code and Documentation. *IEEE Trans. Soft. Eng.*, 28(10), 970-983.
- [14] Baier, C., & Katoen, J. P. (2008). *Principles of Model Checking*: MIT Press.
- [15] Baresi, L., Heckel, R., Thone, S., & Varro, D. (2003). Modeling and Validation of Service-Oriented Architectures: Application vs. Style. *ESEC/FSE'03, ACM*, 68-77.

- [16] Bas, L., Clements, P., & Kazman, R. (1998). *Software Architecture in Practice*: Addison-Wesley.
- [17] Baudry, B., Nebut, C., & Le Traon, Y. (2007). Model-driven engineering for requirements analysis. *EDOC 2007*, 459-466.
- [18] Benammar, M., & Belala, F. (2010). How to Make AADL Specification More Precise. *International Journal of Computer Applications*, 8(10), 16-23.
- [19] Benammar, M., Belala, F., & Latreche, F. (2008). AADL Behavioral Annex based on Generalized Rewriting Logic. *RCIS 2008*.
- [20] Berthomieu, B., Bodeveix, J. P., Farail, P., Filali, M., Garavel, H., Gaufillet, P., et al. (2008). Fiacre: an Intermediate Language for Model Verification in the TOPCASED Environment. *4th European Congress on Embedded Real-Time Software, ERTS 2008*.
- [21] Berthomieu, B., Bodeveix, J. P. C., C., Dal-Zilio, S., Filali, M., & Vernadat, F. (2009). Formal Verification of AADL Specifications in the Topcased Environment. *Ada-Europe'09, LNCS(5570)*.
- [22] Bohner, S. A. (2002). Extending Software Change Impact Analysis into COTS Components. *27th Annual NASA Goddard Software Engineering Workshop*, 175-182.
- [23] Bohner, S. A. (2002). Software Change Impacts – An Evolving Perspective. *ICSM'02*, 263-271.
- [24] Bohner, S. A., & Arnold, R. S. (1996). Software Change Impact Analysis. *IEEE Computer Society Press*.
- [25] Bohner, S. A., & Gracanin, D. (2003). Software Impact Analysis in a Virtual Environment. *28th Annual NASA Goddard Software Engineering Workshop*, 143-151.
- [26] Bonde, L., Boulet, P., & Dekeyser, J. L. (2005). *Traceability and Interoperability at Different Levels of Abstraction in Model-Driven Engineering*. Paper presented at the FDL 2005.
- [27] Borland Caliber Analyst.
- [28] Boronat, A., & Mesequer, J. (2010). An Algebraic Semantics for MOF. *Formal Aspects of Computing*, 22, 269-296.
- [29] Boronat, A., & Mesequer, J. (2009). Algebraic Semantics of OCL-constrained Metamodel Specification. *TOOLS, LNBP(47)*, 96-115.
- [30] Boronat, A., & Ölveczky, P. C. (2010). Formal Real-Time Model Transformations in MOMENT2. *FASE 2010, LNCS(6013)*, 29-43.
- [31] Bose, P. K. (1999). Automated Translation of UML Models of Architectures for Verification and Simulation Using SPIN. In: *ASE'99*.
- [32] Boudiaf, N., Mokhati, F., & Badri, M. (2008). Supporting Formal Verification of DIMA Multi-Agents Models: Towards a Framework Based on Maude Model Checking. *Int. J. Soft. Eng. Knowl. Eng.*, 18(7), 853-875.
- [33] Bozzano, M., Cimatti, A., Katoen, J. P., Nguyen, V. N., Noll, T., & Roveri, M. (2010). Safety, Dependability, and Performance Analysis of Extended AADL Models. *The Computer Journal*.
- [34] Bozzano, M., Cimatti, A., Roveri, M., Katoen, J. P., Nguyen, V. N., & Noll, T. (2009). Verification and Performance Evaluation of AADL Models. *ESEC-FSE'09, ACM*, 285-286.
- [35] Briand, L. C., Labiche, Y., & O'Sullivan, L. (2003). Impact Analysis and Change Management of UML Models. *International Conference on Software Maintenance*, 256-265.
- [36] Briand, L. C., Labiche, Y., O'Sullivan, L., & Sowka, M. (2006). Automated Impact Analysis of UML Models. *Journal of Systems and Software*, 79(3), 339-352.
- [37] Brottier, E., Baudry, B., Le Traon, Y., Touzet, D., & Nicolas, B. (2007). Producing a global requirement model from multiple requirement specifications. *EDOC 2007*, 390-404.

- [38] Bruni, R., Buccharone, A., Gnesi, S., & Melgratti, H. (2008). Modelling Dynamic Software Architectures using Typed Graph Grammars. *GT-VC 2007 ENTCS*, 213(1), 39-53.
- [39] Buccharone, A., & Galeotti, J. P. (2008). Dynamic Software Architectures Verification using DynAlloy. *GT-VMT 2008, Proceedings of the Seventh International Workshop on Graph Transformation and Visual Modeling Techniques*.
- [40] Buckley, J., Mens, T., Zenger, M., Rashid, A., & Kniesel, G. (2005). Towards a Taxonomy of Software Change. *Journal of Software Maintenance and Evolution: Research and Practice*, 17, 309-332.
- [41] Buhr, R. J. A. (1998). Use Case Maps as Architectural Entities for Complex Systems. *IEEE Trans. Softw. Eng.*, 24(12), 1131-1155.
- [42] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., & Stal, M. (1996). *Pattern-Oriented Software Architecture, A System of Patterns*: Wiley.
- [43] Carlshamre, P., Sandahl, K., Lindvall, M., Regnell, B., & Natt och Dag, J. (2001). An industrial survey of requirements interdependencies in software product release planning. *Proceedings of the 5th International Symposium on Requirements Engineering*, 84-91.
- [44] Ceron, R., Duenas, J. C., Serrano, E., & Capilla, R. (2005). A meta-model for requirements engineering in system family context for software process improvement using CMMI. *PROFES 2005*, 3547, 173-178.
- [45] Cheng, H., Xia, Y., & Hu, X. (2007). Requirements Change Management of Information System Based on the Keyword Mapping. *The Sixth Wuhan International Conference on E-Business*, 135-140.
- [46] Chkouri, M. Y., Robert, A., Bozga, M., & Sifakis, J. (2009). Translating AADL into BIP - Application to the Verification of Real-time Systems. *Models in Software Engineering: Workshops and Symposia at MODELS 2008, LNCS(5421)*, 5-19.
- [47] Ciraci, S. (2009). *Graph based Verification of Software Evolution Requirements* (Vol. PhD thesis 09-162): Univ. of Twente.
- [48] Clavel, M., Duran, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., et al. (2002). Maude: Specification and Programming in Rewriting Logic. *Theoretical Computer Science*, 285, 187-243.
- [49] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., et al. (2007). All about Maude - A High-Performance Logical Framework. *Lecture Notes in Computer Science*, 4350.
- [50] Cleland-Huang, J., Chang, C. K., & Christensen, M. (2003). Event-based Traceability for Managing Evolutionary Change. *IEEE Transactions on Software Engineering*, 29(9), 796-810.
- [51] Cleland-Huang, J., & Schmelzer, D. (2003). Dynamically Tracing Non-Functional Requirements through Design Pattern Invariants. *In Proceedings of the Second International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE'03)*.
- [52] Cleland-Huang, J., Settimi, R., BenKhadra, O., Berezhanskaya, E., & Christina, S. (2005). Goal-centric Traceability for Managing Non-functional Requirements. *In Proceedings of the 27th International Conference on Software Engineering (ICSE'05)*, 362-371.
- [53] Clements, P., Kazman, R., & Klein, M. (2002). *Evaluating Software Architectures: Methods and Case Studies*: Addison-Wesley Professional.
- [54] Cockburn, A. (2000). Writing Effective Use Cases. *Addison-Wesley*.
- [55] COMET (Component and Model Based Development Methodology). from <http://modelbased.net/methods/comet/>
- [56] Compare, D., Inverardi, P., & Wolf, A. L. (1999). Uncovering Architectural Mismatch in Component Behavior. *Sci. Comput. Prog.*, 33(2), 101-131.

- [57] Corradini, F., & Inverardi, P. (1998). Model Checking Cham Description of Software Architecture. *WICSA'98*.
- [58] Cysneiros, G., & Zisman, A. (2008). Traceability and Completeness Checking for Agent-Oriented Systems. *SAC 2008*, 71-77.
- [59] Dahlstedt, A. G., & Persson, A. (2005). Requirements Interdependencies: State of the Art and Future Challenges. In A. Aurum & C. Wohlin (Eds.), *Engineering and Managing Software Requirements* (pp. 95-116). Berlin: Springer.
- [60] Dashofy, E. M., Hoek, A., & Taylor, R. N. (2005). A Comprehensive Approach for the Development of Modular Software Architecture Description Languages. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 14(2), 199-245.
- [61] de Niz, D., & Feiler, P. H. (2009). Verification of Replication Architectures in AADL. *14th International Conference on Engineering of Complex Computer Systems, IEEE Computer Society*, 365-370.
- [62] Dean, M., Schreiber, G., van Harmelen, F., Hendler, J., Horrocks, I., McGuinness, D., et al. OWL Web Ontology Language Reference W3C Recommendation (2004).
- [63] Delanote, D., van Baelen, S., Joosen, W., & Berbers, Y. (2007). Using AADL in Model Driven Development. *IEEE-SEE International Workshop on UML and AADL 2007, International Conference on Engineering Complex Computer Systems (ICECCS07)*.
- [64] Dias, M. S., & Vieira, M. E. R. (2000). Software Architecture Analysis based on Statechart Semantics. *IWSSD'00, IEEE Computer Society*, 133-138.
- [65] Diskin, Z., Xiong, Y., & Czarnecki, K. (2010). From State- to Delta-Based Bidirectional Model Transformations. *ICMT 2010, LNCS(6142)*, 61-76.
- [66] Drivalos, N., Kolovos, D. S., Paige, R. F., & Fernandes, K. J. (2009). Engineering a DSL for Software Traceability. *SLE 2008, LNCS (5452)*, 151-167.
- [67] Duffy, D., MacNish, C., McDermid, J., & Morris, P. (1995). A framework for requirements analysis using automated reasoning. *CAiSE 1995, Lecture Notes in Computer Science*, 932, 68-81.
- [68] Edwards, G., Malek, S., & Medvidovic, N. (2007). Scenario-Driven Dynamic Analysis of Distributed Architectures. *10th International Conference on Fundamental Approaches to Software Engineering (FASE'07)*, 125-139.
- [69] Egyed, A. (2000). Automatically Validating Model Consistency during Refinement.
- [70] Egyed, A. (2003). A Scenario-Driven Approach to Trace Dependency Analysis. *IEEE Trans. Software Eng.*, 29(2), 116-132.
- [71] Egyed, A., & Grunbacher, P. (2002). Automated Requirements Traceability: beyond the Record and Replay Paradigm. *17th IEEE International Conference on Automated Software Engineering (ASE'02)*, 163-171.
- [72] Egyed, A., & Grunbacher, P. (2005). Supporting Software Understanding with Automated Requirements Traceability. *Int. J. Soft. Eng. Knowl. Eng.*, 15(5), 783-810.
- [73] Egyed, A., & Wile, D. (2001). Statechart Simulator for Modeling Architecture Dynamics. *WICSA'01, IEEE Computer Society*, 87-96.
- [74] Erlikh, E. (2000). Leveraging Legacy System Dollars for E-Business. *IT Professional*, 2(3), 17-23.
- [75] Escalona, M. J., & Aragon, G. (2008). NDT. A Model-Driven Approach for Web Requirements. *IEEE Trans. Soft. Eng.*, 34(3), 377-390.
- [76] Falkenberg, E. D., Hesse, W., Lindgreen, P., Nilsson, B. E., Han Oei, J. E., Rolland, C., et al. (1998). *A Framework of Information System Concepts*.

- [77] Falleri, J., Huchard, M., & Nebut, C. (2006). Towards a Traceability Framework for Model Transformations in Kermeta. In *Traceability Workshop, at European Conference on Model Driven Architecture (ECMDA-TW 2006)*, 31-40.
- [78] Fawcett, T. (2004). ROC Graphs: Notes and Practical Considerations for Researchers. *Technical Report, HP Laboratories, Palo Alto, California*.
- [79] Feiler, P. H., Lewis, B., & Vestal, S. (2003). The SAE Avionics Architecture Description Language (AADL) Standard: A Basis for Model-Based Architecture-Driven Embedded Systems Engineering. *RTAS 2003 Workshop on Model-Driven Embedded Systems*.
- [80] Feng, T., & Maletic, J. I. (2006). Applying Dynamic Change Impact Analysis in Component-based Architecture Design. *Seventh ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD'06)*.
- [81] Finkelstein, A. C. W., Gabbay, D., Hunter, A., Kramer, J., & Nuseibeh, B. (1994). Inconsistency handling in multiperspective specifications. *IEEE Transactions on Software Engineering*, 20(8), 569-578.
- [82] Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code*: Addison-Wesley.
- [83] Franca, R., Bodeveix, J. P., Filali, M., Rolland, J. F., Chemouil, D., & Thomas, D. (2007). The AADL Behaviour Annex - Experiments and Roadmap. *ICECCS 07*.
- [84] FST. (1992). Failures Divergence Refinement: User Manual and Tutorial.
- [85] Gallagher, K. B., & Lyle, J. R. (1991). Using Program Slicing in Software Maintenance. *IEEE Transactions on Software Engineering*, 17(8), 751-761.
- [86] Galster, M., Eberlein, A., & Moussavi, M. (2006). Transition from Requirements to Architecture: A Review and Future Perspective. *SNPD '06*.
- [87] Galvao, I., & Goknil, A. (2007). Survey of Traceability Approaches in Model-Driven Engineering. *EDOC'07*, 313-324.
- [88] Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*: Addison-Wesley Professional.
- [89] Garlan, D., Allen, R., & Ockerbloem, J. (1995). Architectural Mismatch or Why It's Hard to Build Systems Out of Existing Parts. *17th International Conference on Software Engineering (ICSE'95)*, 179 - 185.
- [90] Garlan, D., Monroe, R. T., & Wile, D. (1997). ACME: An Architecture Description Language. *CASCON'97*, 169-183.
- [91] Gilles, O., & Hugues, J. (2010). Expressing and Enforcing User-defined Constraints of AADL Models. *Proceedings of the 5th UML\& AADL Workshop (UML\&AADL 2010)*.
- [92] Gilles, O., & Hugues, J. (2008). Validating Requirements at Model-Level. *Proceedings of the 4th workshop on Model-Oriented Engineering (IDM'08)*.
- [93] Giorgini, P., Mylopoulos, J., Nicchiarelli, E., & Sebastiani, R. (2003). Formal reasoning techniques for goal models. *Journal on Data Semantics, Lecture Notes in Computer Science*, 2800, 1-20.
- [94] Goknil, A. (2009). Tutorial: requirements relations and definitions with examples. from http://www.home.cs.utwente.nl/~goknila/tutorial/Relations_Tutorial.doc
- [95] Goknil, A., Kurtev, I., & van den Berg, K. (2008). Change Impact Analysis based on Formalizations of Trace Relations for Requirements. *ECMDA-TW'08, SINTEF Report*, 59-75.
- [96] Goknil, A., Kurtev, I., & van den Berg, K. (2008). A Metamodeling Approach for Reasoning about Requirements. *European Conference on Model Driven Architecture Foundations and Applications (ECMDA-FA'08), LNCS(5095)*, 311-326.

- [97] Goknil, A., Kurtev, I., & van den Berg, K. (2010). Tool Support for Generation and Validation of Traces between Requirements and Architecture. *ECMFA-TW 2010*.
- [98] Goknil, A., Kurtev, I., van den Berg, K., & Veldhuis, J. W. (2011). Semantics of Trace Relations in Requirements Models for Consistency Checking and Inferencing. *Software and System Modeling*, 10(1), 31-54.
- [99] Gorlick, M. M., & Razouk, R. R. (1991). Using Weaves for Software Construction and Analysis. *Thirteenth International Conference on Software Engineering*, 23-34.
- [100] Gotel, O. C. Z., & Finkelstein, C. W. (1994). An Analysis of the Requirements Traceability Problem. *RE'94*, 94-101.
- [101] Graph Visualization Software (Graphviz). from <http://www.graphviz.org>
- [102] Grechanik, M., McKinley, K. S., & Perry, D. E. (2007). Recovering and Using use-case-diagram-to-source-code traceability links. *ESEX-FSE '07*.
- [103] Grunbacher, P., Egyed, A., & Medvidovic, N. (2003). Reconciling Software Requirements and Architectures with Intermediate Models. *Softw. Syst. Modeling*, 3, 235-253.
- [104] Gui, S., Luo, L., Li, Y., & Wang, L. (2008). Formal Schedulability Analysis and Simulation for AADL. *ICESS 2008*.
- [105] Hall, J. G., Jackson, M., Laney, R. C., Nuseibeh, B., & Rapanotti, L. (2002). Relating Software Requirements and Architectures using Problem Frames. *RE '02*, 137-144.
- [106] Han, J. (1997). Supporting Impact Analysis and Change Propagation in Software Engineering Environments. *Proceedings of the 8th International Workshop on Software Technology and Engineering Practice (STEP '97)*, 172-182.
- [107] Harker, S. D. P., Eason, K. D., & Dobson, J. E. (1993). The Change and Evolution of Requirements as a Challenge to the Practice of Software Engineering. *Proceedings of IEEE International Symposium on Requirements Engineering 1993*, 266-272.
- [108] Hassine, J., Rilling, J., & Hewitt, J. (2005). Change Impact Analysis for Requirement Evolution using Use Case Maps. *Eighth International Workshop on Principles of Software Evolution*, 81-90.
- [109] Hayes, J. H., Dekkyar, A., & Sundaram, S. K. (2006). Advancing Candidate Link Generation for Requirements Tracing: the Study of Methods. *IEEE Trans. Softw. Eng.*, 32(1), 4-19.
- [110] Hearnden, D., Lawley, M., & Raymond, K. (2006). Incremental Model Transformation for the Evolution of Model-Driven Systems. *In MoDELS 2006*, 321-335.
- [111] Heaven, W., & Finkelstein, A. (2004). UML profile to support requirements engineering with KAOS. *IEE Proceedings Software*, 151(1), 10-27.
- [112] Heckel, R., & Thone, S. (2005). Behavior-Preserving Refinement Relations between Dynamic Software Architectures. *WADT'04, LNCS(3423)*, 1-27.
- [113] Heitmeyer, C. L., Jeffords, R. D., & Labaw, G. L. (1996). Automated consistency checking of requirements specifications. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 5(3), 231-261.
- [114] Heninger, K. L. (1980). Specifying Software Requirements for Complex Systems: New Techniques and Their Application. *IEEE Trans. Soft. Eng.*, 6(1), 2-13.
- [115] Honeywell. (1998). MetaH Evaluation and Support Site. from <http://www.htc.honeywell.com>
- [116] Hugues, J., Zalila, B., Pautet, L., & Kordon, F. (2008). From the Prototype to the Final Embedded System using the Ocarina AADL Tool Suite. *ACM Trans. Embedded Comput. Syst.*, 7(4).

- [117] Hunter, A., & Nuseibeh, B. (1998). Managing inconsistent specifications: reasoning, analysis, and action. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 7(4), 335-367.
- [118] Huth, M. R. A., & Ryan, M. D. (2000). Logic in Computer Science: Modeling and Reasoning about Systems. *Cambridge University Press, Cambridge*.
- [119] IBM Rational RequisitePro. from <http://www-01.ibm.com/software/awdtools/reqpro/>
- [120] IBM Telelogic Doors. from <http://www.telelogic.com/Products/doors/index.cfm>
- [121] Ibrahim, S., Munro, M., & Deraman, A. (2005). A Requirements Traceability to Support Change Impact Analysis. *Asian Journal of Information Technology*, 4(4), 329-338.
- [122] IEEE. (1984). IEEE Guide to Software Requirements Specification.
- [123] IEEE. (1990). IEEE Standard Glossary of Software Engineering Terminology.
- [124] INCOSE Requirements Management Tool Survey. from <http://www.incos.org>
- [125] Inverardi, P., & Wolf, A. L. (1995). Formal Specification and Analysis of Software Architectures Using the Chemical Abstract Machine Model. *IEEE Trans. Softw. Eng.*, 21(4), 373-386.
- [126] Jackson, D. (2002). Alloy: a Lightweight Object Modelling Notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2), 256-290.
- [127] Jahier, E., Halbwachs, N., Raymond, P., Nicollin, X., & Lesens, D. (2007). Virtual Execution of AADL Models via a Translation into Synchronous Programs. *EMSOFT'07, ACM*, 134-143.
- [128] Jahier, E., Raymond, P., & Baufreton, P. (2006). Case studies with Lurette V2. *International Journal on Software Tools for Technology Transfer (STTT)*, 8(6), 517-530.
- [129] Java. from <http://java.sun.com/>
- [130] Jena. A Semantic Web Framework for JAVA. from <http://jena.sourceforge.net/>
- [131] JGraph. Java Graph Visualization and Layout. from <http://www.jgraph.com/>
- [132] Johann, S., & Egyed, A. (2004). Instant and Incremental Transformation of Models. *19th IEEE International Conference on Automated Software Engineering (ASE'04)*, 362-365.
- [133] Jonsson, P., & Lindvall, M. (2005). Impact Analysis. In A. Aurum & C. Wohlin (Eds.), *Engineering and Managing Software Requirements* (pp. 117-142). Berlin: Springer.
- [134] Jouault, F. (2005). Loosely Coupled Traceability for ATL. In *Traceability Workshop at European Conference on Model Driven Architecture (ECMDA-TW 2005)*, 29-37.
- [135] Jouault, F., Allilaire, F., Bezivin, J., & Kurtev, I. (2008). ATL: A Model Transformation Tool. *Sci. Comput. Prog.*, 72(1-2), 31-39.
- [136] Jouault, F., & Kurtev, I. (2006). Transforming Models with ATL. *MoDELS 2005, LNCS(3844)*.
- [137] Kaindl, H. (1999). Difficulties in the Transition from OO Analysis to Design. *IEEE Software*, 16, 94-102.
- [138] Khan, S. S., Greenwood, P., Garcia, A., & Rashid, A. (2008). On the Impact of Evolving Requirements-Architecture Dependencies: An Exploratory Study. *Caise 2008, LNCS(5074)*, 243-257.
- [139] Kilpinen, M. S. (2008). *The Emergence of Change at the System Engineering and Software Design Interface: An Investigation of Impact Analysis*. PhD Thesis, University of Cambridge, Cambridge.
- [140] Kitchenham, B. A., Travassos, G. H., von Mayrhauser, A., Niessink, F., Schneidewind, N. F., Singer, J., et al. (1999). Towards an Ontology of Software Maintenance. *Journal of Software Maintenance: Research and Practice*, 11, 365-389.
- [141] Klusener, A. S., Lammel, L. R., & Verhoef, C. (2005). Architectural Modifications to Deployed Software. *Science of Computer Programming*, 54, 143-211.

- [142] Knethen, A. v., & Paech, B. (2002). A Survey on Tracing Approaches in Practice and Research. *IESE-Report, 095.01/E*(version 1.0).
- [143] Koch, N., & Kraus, A. (2003). Towards a common metamodel for the development of web applications. *ICWE 2003*, 497-506.
- [144] Kolovos, D., Paige, R., & Polack, F. (2006). Merging Models with the Epsilon Merging Language. In *Proceedings of ACM/IEEE 9th International Conference on Model Driven Engineering Languages and Systems (Models/UML'06)*.
- [145] Kolovos, D., Paige, R., & Polack, F. (2006). On-Demand Merging of Traceability Links with Models. *Traceability Workshop at European Conference on Model Driven Architecture (ECMDA-TW 2006)*.
- [146] Kurtev, I., Dee, M., Goknil, A., & van den Berg, K. (2007). Traceability-based Change Management in Operational Mappings. *Traceability Workshop at European Conference on Model Driven Architecture (ECMDA-TW 2007)*.
- [147] Lago, P., Muccini, H., & van Vliet, H. (2009). A Scoped Approach to Traceability Management. *The Journal of Systems and Software*, 82, 168-182.
- [148] Lai, W. (2009). *Relationship-Based Change Propagation: A Case Study*. M.Sc. thesis, University of Toronto, Toronto.
- [149] Lai, W., Nejati, S., Cabot, J., Diskin, Z., Easterbrook, S., Sabetzadeh, M., et al. (2009). Relationship-Based Change Propagation: A Case Study. In *Proceedings of ICSE'09 Workshop on Modeling in Software Engineering (MiSE'09)*.
- [150] Lam, W., & Shankararaman, V. (1998). Managing Change in Software Development using a Process Improvement Approach. *Proceedings of 24th Euromicro Conference 1998*, 779-786.
- [151] Lamsweerde, A. v. (2009). *Requirements Engineering: From System Goals to UML Models to Software Specifications*. John Wiley & Sons.
- [152] Lee, J., & Kuo, J. Y. (1998). New Approach to Requirements Trade-off Analysis for Complex Systems. *IEEE Transactions on Knowledge Data Engineering*, 10(4), 551-562.
- [153] Lee, M., Offutt, J. A., & Alexander, R. T. (2000). Algorithmic Analysis of the Impacts of Changes to Object-oriented Software. *34th International Conference on Technology of Object-Oriented Languages and Systems*, 61-70.
- [154] Lee, W. T., Deng, W. Y., Lee, J., & Lee, S. J. (2010). Change Impact Analysis with a Goal-Driven Traceability-Based Approach. *International Journal of Intelligent Systems*.
- [155] Li, C., Zhou, X., & Dong, Y. (2010). Formal Behavior Specification for AADL. *2nd International Conference on Industrial and Information Systems (IIS)*, 110-113.
- [156] Limon, A. E., & Garbajosa, J. (2005). The Need for a Unifying Traceability Scheme. In *Traceability Workshop at European Conference on Model Driven Architecture (ECMDA-TW 2005)*, 47-55.
- [157] Liu, W. (2004). Architecting Requirements. *Doctoral Consortium at RE '04*.
- [158] Liu, W., & Easterbrook, S. (2003). Eliciting Architectural Decisions from Requirements using a Rule-based Framework. *STRAW '03*.
- [159] Lock, S. (2001). *A Hybrid Approach to Requirement Level Impact Analysis*. PhD Thesis, Lancaster University.
- [160] Lock, S., & Kotonya, G. (1999). An Integrated Framework for Requirement Change Impact Analysis. *Proceedings of the 4th Australian Conference on Requirements Engineering*, 29-42.
- [161] Lock, S., & Kotonya, G. (1999). An Integrated, Probabilistic Framework for Requirement Change Impact Analysis. *The Australian Journal of Information Systems*, 6(2), 38-63.

- [162] Loniewski, G., Insfran, E., & Abrahao, S. (2010). A Systematic Review of the Use of Requirements Engineering Techniques in Model-Driven Development. *MODELS 2010, LNCS(6395)*, 213-227.
- [163] Looman, S. A. M. (2009). Impact Analysis of Changes in Functional Requirements in the Behavioral View of Software Architectures. *M.Sc. Thesis, University of Twente*.
- [164] Lopez, O., Laguna, M. A., & Garcia, F. J. (2002). Metamodeling for requirements reuse. *Anais do WER02—Workshop em Engenharia de Requisitos*, 76-90.
- [165] Luckham, D. C., & Henke, F. W. (1995). Specification and Analysis of System Architecture using Rapide. *IEEE Trans. Soft. Eng.*, 21(4), 336-355.
- [166] Mader, P., & Cleland-Huang, J. (2010). A Visual Traceability Modeling Language. *MODELS 2010, LNCS(6394)*, 226-240.
- [167] Mader, P., O., G., & Philippow, I. (2009). Enabling Automated Traceability Maintenance through the Upkeep of Traceability Relations. *ECMDA-FA '09, LNCS(5562)*, 174-189.
- [168] Magee, J. (1999). Behavioral Analysis of Software Architectures using LTSA. *ICSE'99*, 634-637.
- [169] Magee, J., Dulay, N., Eisenbach, S., & Kramer, J. (1995). Specifying Distributed Software Architectures. *Fifth European Software Engineering Conference (ESEC 95)*, 137-153.
- [170] Magee, J., & Kramer, J. (1996). Dynamic Structure in Software Architectures. *ACM SIGSOFT Software Engineering Notes*, 21(6), 3 - 14.
- [171] Magee, J., Kramer, J., & Giannakopoulou, D. (1999). Behaviour Analysis of Software Architectures. *First Working IFIP Conference on Software Architecture (WICSA1), IFIP Conference Proceedings(140)*, 35-50.
- [172] Maier, M. W., Emery, D., & Hilliard, R. (2001). Software Architecture: Introducing IEEE Standard 1471. *IEEE Computer*, 34(4), 107-109.
- [173] Malan, R., & Bredemeyer, D. (2002). Architectural Requirements in the Visual Architecting Process. from <http://www.bredemeyer.com/ArchitectingProcess/ArchitecturalRequirements.htm>
- [174] McCormack, A., Rusnak, J., & Baldwin, C. Y. (2006). Exploring the Structure of Complex Software Designs: An Empirical Study of Open Source and Proprietary Code. *Management Science*, 52(7), 1015-1030.
- [175] Medvidovic, N., & Taylor, R. N. (2000). A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Trans. Soft. Eng.*, 26(1), 70-93.
- [176] The Metamodeling Language Kermeta. from <http://www.kermeta.org>
- [177] Meyer, J. J. C., Wieringa, R., & Dignum, F. (1998). The Role of Deontic Logic in the Specification of Information Systems. *Logics for Databases and Information Systems*, 71-115.
- [178] Molina, F., Pardillo, J., Cachero, C., & Toval, A. (2010). An MDE Modeling Framework for Measurable Goal-Oriented Requirements. *International Journal of Intelligent Systems*, 25(8), 757-783.
- [179] Molina, F., Pardillo, J., Cachero, C., & Toval, A. (2009). A Systematic Review of Requirements Metamodels. *Technical Report, University of Murcia*.
- [180] Molina, F., Pardillo, J., & Toval, A. (2008). Modelling Web-Based Systems Requirements Using WRM. *Web Information Systems Engineering – WISE 2008 Workshops, LNCS(5176)*, 122-131.
- [181] Molina, F., & Toval, A. (2009). Integrating Usability Requirements that can be Evaluated in Design Time into Model Driven Engineering of Web Information Systems. *Advances in Engineering Software*, 40(12), 1306-1317.

- [182] Moment2-AADL. from <http://www.cs.le.ac.uk/people/aboronat/tools/moment2-aadl/>
- [183] Moon, M., Yeom, K., & Chae, H. S. (2005). An approach to developing domain requirements reuse as a core asset based on commonality and variability analysis in a product line. *IEEE Transactions on Software Engineering*, 31(7), 551-569.
- [184] Moriconi, M., Qian, X., & Riemenschneider, R. A. (1995). Correct Architecture Refinement. *IEEE Trans. Softw. Eng.*, 21(4), 356-372.
- [185] Murphy, G. C., Notkin, D., & Sullivan, K. (1995). Software Reflexion Models: Bridging the Gap between Source and High-Level Models. *SIGSOFT'95*.
- [186] Mylopoulos, J., Chung, L., & Yu, E. (1999). From object-oriented to goal oriented requirements analysis. *ACM Communications*, 31-37.
- [187] Naslavsky, L., Xu, L., Dias, M., Ziv, H., & Richardson, D. J. (2004). Extending xADL with Statechart Behavioral Specification. *WADS '04 at ICSE 2004*.
- [188] Naumovich, G., G.S., A., Clarke, L. A., & Osterweil, L. J. (1997). Applying Static Analysis to Software Architectures. *ESEC/FSE'97, LNCS(1301)*, 77-93.
- [189] Navarro, E., Mocholi, J. A., Letelier, P., & Ramos, I. (2006). A metamodeling approach for requirements specification. *The Journal of Computer Information Systems* 46(5), 67-77.
- [190] .Net Platform. from <http://msdn.microsoft.com/en-gb/netframework/default.aspx>
- [191] Noppen, J., van den Broek, P., & Aksit, M. (2007). Imperfect Requirements in Software Development. *REFSQ 2007*, 4542, 247-261.
- [192] Nurmuliani, N., Zowghi, D., & Williams, S. P. (2004). Using Card Sorting Technique to Classify Requirements Change. *Proceedings of 12th IEEE International Requirements Engineering Conference 2004*, 240-248.
- [193] Nuseibeh, B. (2001). Weaving together Requirements and Architecture. *IEEE Software*, 34, 115-111.
- [194] Nuseibeh, B., Kramer, J., & Finkelstein, A. (1994). A framework for expressing the relationships between multiple views in requirements specification. *IEEE Transactions on Software Engineering*, 760-773.
- [195] O'Neal, J. S. (2003). *Analyzing the Impact of Changing Software Requirements: A traceability-based Methodology*. Ph.D. dissertation, Louisiana State University.
- [196] O'Neal, J. S., & Carver, D. L. (2001). Analyzing the Impact of Changing Requirements. *International Conference on Software Maintenance*, 190-195.
- [197] Ölveczky, P. C., Boronat, A., & Mesequer, J. (2010). Formal Semantics and Analysis of Behavioral AADL Models in Real-Time Maude. *FMOODS/FORTE 2010, LNCS(6117)*, 47-62.
- [198] Ölveczky, P. C., Boronat, A., Mesequer, J., & Pek, E. (to appear). Formal Semantics and Analysis of Behavioral AADL Models in Real-Time Maude. *Technical Report at UIUC*.
- [199] OMG. MDA Guide. from <http://www.omg.org/cgi-bin/doc?omg/03-06-01>
- [200] OMG. SysML Specification. Retrieved 05 January 2010, from
<http://www.sysml.org/specs.htm>
- [201] OMG. (2004). UML 2.0 Superstructure Specification. from
<http://www.omg.org/cgi-bin/doc?ptc/2004-10-02>.
- [202] OMG Semantics of Business Vocabulary and Rules (SBVR). *OMG Standard*.
- [203] Ommering, R., Linden, F., Kramer, J., & Magee, J. (2000). The Koala Component Model for Consumer Electronics Software. *IEEE Computer*, 33(3), 78-85.
- [204] The Open Source Toolkit for Critical Systems (Topcased).

- [205] Oquendo, F. (2004). π -ARL: an Architecture Refinement Language for Formally Modelling the Stepwise Refinement of Software Architectures. *ACM SIGSOFT Software Engineering Notes*, 29(5), 1-20.
- [206] Paige, R. F., Drivalos, N., Kolovos, D. S., Fernandes, K. J., Power, C., Olsen, G. K., et al. (2010). Rigorous Identification and Encoding of Trace-links in Model-Driven Engineering. *Software and Systems Modeling*.
- [207] Paige, R. F., Kolovos, D. S., & Polack, F. A. C. (2005). Refinement via Consistency Checking in MDA. *ENTCS*, 137(2), 151-161.
- [208] Paige, R. F., Olsen, G. K., Kolovos, D. S., Zschaler, S., & Power, C. (2008). Building Model-Driven Engineering Traceability Classifications. In *Proceedings of ECMDA Traceability Workshop (ECMDA-TW 2008)*, 49-58.
- [209] Pelliccione, P., Inverardi, P., & Muccini, H. (2009). CHARMY: A Framework for Designing and Verifying Architectural Specifications. *IEEE Trans. Software Eng.*, 35(3), 325-346.
- [210] Perry, D., & Wolf, A. L. (1992). Foundations for the Study of Software Architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4), 40-52.
- [211] Pinheiro, F. A. C. (2003). Requirements traceability *Perspectives on Software Requirements* (pp. 93-113): Springer.
- [212] Post, H., Sinz, C., Merz, F., Gorges, T., & Kropf, T. (2009). Linking Functional Requirements and Software Verification. *RE'09*, 295-302.
- [213] QuadREAD. (2006). Quality-Driven Requirements Engineering and Architectural Design. 2010, from <http://quadread.ewi.utwente.nl/>
- [214] Ramesh, B., & Edwards, M. (1993). Issues in the Development of a Requirements Traceability Model. In *Proceedings of the IEEE International Symposium on Requirements Engineering*, 256-259.
- [215] Ramesh, B., & Jarke, M. (2001). Towards reference Models for Requirements Traceability. *IEEE Trans. Softw. Eng.*, 27(1), 58-93.
- [216] Rashid, A., Moreira, A., & Araujo, J. (2003). Modularization and composition of aspectual requirements. *AOSD 2003*, 11-20.
- [217] Ratel, C., Halbwachs, N., & Raymond, P. (1991). Programming and verifying critical systems by means of the synchronous data-flow programming language lustre. *ACM-SIGSOFT'91 Conference on Software for Critical Systems*.
- [218] RCP. Eclipse Rich Client Platform. from <http://www.eclipse.org/home/categories/rcp.php>
- [219] Rensink, A. (2003). The GROOVE Simulator: A Tool for State Space Generation. *Proceedings of Applications of Graph Transformations with Industrial Relevance (AGTIVE) - LNCS*, 3062, 479-485.
- [220] RIF. Requirements Interchange Format. from <http://www.automotive-his.de/rif/doku.php>
- [221] Rivera, E. J., Guerra, E., de Lara, J., & Vallecillo, A. (2009). Analyzing Rule-Based Behavioral Semantics of Visual Modeling Languages with Maude. *SLE 2008, LNCS* (5452), 54-73.
- [222] Robinson, W. N., Pawłowski, S. D., & Volkov, V. (2003). Requirements interaction management. *ACM Computing Surveys*, 35(2), 132-190.
- [223] Rodrigues, O., Garcez, A., & Russo, A. (2004). Reasoning about requirements evolution using clustered belief revision. *SBLA 2004, Lecture Notes in Computer Science (LNAAI)*, 3171, 41-51.
- [224] Sabaliauskaite, G., Loconsole, A., Engstrom, E., Unterkalmsteiner, M., Regnell, B., Runeson, P., et al. (2010). Challenges in Aligning Requirements Engineering and Verification in a Large-Scale Industrial Context. *REFSQ 2010, LNCS*(6182), 128-142.

- [225] SAE. Architecture Analysis and Design Language (AADL). Retrieved 05 January 2010, from <http://www.aadl.info>
- [226] Sanchez, O., Molina, F., Garcia-Molina, J., & Toval, A. (2009). ModelSec: A Generative Architecture for Model-Driven Security. *Journal of Universal Computer Science*, 15(15), 2957-2980.
- [227] Schmidt, D. C. (2006). Guest Editor's Introduction: Model-Driven Engineering. *IEEE Computer*, 39(2), 25-31.
- [228] Schwarz, H., Ebert, J., & Winter, A. (2009). Graph-based Traceability: a Comprehensive Approach. *Software and System Modeling*.
- [229] SEI Software Architecture Glossary. from
<http://www.sei.cmu.edu/architecture/start/glossary/>
- [230] Seidewitz, E. (2003). What Models Mean. *IEEE Software*, 20(5).
- [231] Soares, M. S., & Vrancken, J. (2008). Model-driven user requirements specification using SysML. *Journal of Software*, 3(6), 57-68.
- [232] Sokolsky, O., Lee, I., & Clarke, D. (2009). Process-Algebraic Interpretation of AADL Models. *Reliable Software Technologies - Ada Europe, LNCS*(5570), 222-236.
- [233] Sommerville, I. (2001). *Software Engineering* (6 ed.): Addison-Wesley.
- [234] Spencer, J. (2000). Architecture Description Markup Language (ADML): Creating an Open Market for IT Architecture Tools. *The Open Group*.
- [235] Spijkerman, W. (2010). *Tool Support for Change Impact Analysis in Requirement Models*. MSc Thesis, University of Twente, Enschede.
- [236] Supakkul, S., & Chung, L. (2005). A UML profile for goal-oriented and use case driven representation of NFRs and FRs. *SERA 2005*, 112-119.
- [237] Swanson, E. B. (1976). The Dimensions of Maintenance. *Proceedings of the 2nd International Conference on Software Engineering*, 492-497.
- [238] Swanson, E. B., & Chapin, N. (1995). Interview with E. Burton Swanson. *Journal of Software Maintenance: Research and Practice*, 7(5), 303-315.
- [239] SWEBOK. Guide to Software Engineering Body of Knowledge. *IEEE Computer Society*.
- [240] System Level Automation Tool for Engineers (SLATE). from <http://www.tdtech.com>
- [241] Tang, A., Jin, Y., Han, J., & Nicholson, A. (2005). Predicting Change Impact in Architecture Design with Bayesian Belief Networks. *5th Working IEEE/IFIP Conference on Software Architecture*, 67-76.
- [242] Taylor, R. N., Medvidovic, N., & Dashofy, E. M. (2010). *Software Architecture: Foundations, Theory, and Practice*: John Wiley & Sons.
- [243] ten Hove, D., Goknil, A., Kurtev, I., van den Berg, K., & de Goede, K. (2009). Change Impact Analysis for SysML Requirements Models based on Semantics of Trace Relations. *ECMDA-TW 2009*, 17-28.
- [244] Tip, F., Jong, D. C., Field, J., & Ramalingam, G. (1996). Slicing Class Hierarchies in C++. *Object-Oriented Programming, Systems, Languages & Applications Conference*, 179-197.
- [245] Tool for Requirements Inferencing and Consistency Checking (TRIC) from
<http://trese.cs.utwente.nl/tric/>
- [246] TopTeam Analyst. from
http://www.technosolutions.com/topteam_requirements_management.html
- [247] Turver, R. J., & Munro, M. (1994). An Early Impact Analysis Technique for Software Maintenance. *Journal of Software Maintenance Research and Practice*, 6(1), 35-52.

- [248] van der Westhuizen, C., & van der Hoek, A. (2002). Understanding and Propagating Architectural Changes. *Proceedings of the IFIP 17th World Computer Congress- TC2 Stream, 3rd IEEE/IFIP Conference on Software Architecture: System Design, Development and Maintenance*, 95-109.
- [249] van Domburg, R. S. A. (2009). *Empirical Validation of Representation and Interpretation of Software Requirements in Requirements Models*. MSc Thesis, University of Twente, Enschede.
- [250] van Lamswerde, A. (2001). Goal-oriented requirements engineering: a roundtrip from research to Practice. *Invited Minitutorial, Proceedings RE'01—5th International Symposium Requirements Engineering*, 249-263.
- [251] van Lamswerde, A., Darimont, R., & Letier, E. (1998). Managing conflicts in goal-driven requirements engineering. *IEEE Transactions on Software Engineering*, 24(11), 908-926.
- [252] Vanhooff, B., & Berbers, Y. (2005). Supporting Modular Transformation Units with Precise Transformation Traceability Metadata. In *Traceability Workshop at European Conference on Model Driven Architecture (ECMDA-TW 2005)*.
- [253] Varona Gomez, R., & Villar, E. (2009). AADS: AADL Simulation and Performance Analysis in SystemC. *Software demonstration at the DATE'09 University Booth*, Nice.
- [254] Veldhuis, J. W. (2009). *Tool support for a metamodeling approach for reasoning about requirements*. MSc Thesis, University of Twente, Enschede.
- [255] Vicente-Chicote, C., Moros, B., & Toval, A. (2007). REMM-Studio: an integrated model-driven environment for requirements specification, validation and formatting. *Journal of Object Technology*, 6(9), 437-454.
- [256] Vogel, R., & Mantell, K. (2006). MDA Adoption for a SME: evolution, not revolution - Phase II. *2nd Workshop on From Code Centric to Model Centric Software Engineering: Practices, Implications and ROI*.
- [257] Warren, I. (1998). *The Renaissance of Legacy Systems*: Springer.
- [258] Wasson, C. S. (2006). *System, Analysis, Design, and Development: Concepts, Principles, and Practices*: John Wiley & Sons.
- [259] Wieringa, R. (2009). Design Science as Nested Problem Solving. *4th International Conference on Design Science Research in Information Systems and Technology*, ACM, 1-12.
- [260] Wieringa, R. (2010). Relevance and Problem Choice in Design Science. *5th International Conference on Global Perspectives on Design Science Research (DESRIST), Lecture Notes in Computer Science (LNCS)(6105)*, 61-76.
- [261] Wieringa, R., Maide, N., & Mead, N. (2006). Requirements Engineering Paper Classification and Evaluation Criteria: a Proposal and a Discussion. *Requirements Engineering Journal*, 11(1), 102-107.
- [262] Winkler, S., & Pilgrim, J. V. (2010). A Survey of Traceability in Requirements Engineering and Model-Driven Development. *Software and Systems Modeling*.
- [263] Yang, Z., Hu, K., Ma, D., & Pi, L. (2009). Towards a Formal Semantics for the AADL Behavior Annex. *DATE'09*, 1166-1171.
- [264] Zhang, P., Muccini, H., & Li, B. (2009). A Classification and Comparison of Model Checking Software Architecture Techniques. *The Journal of Systems and Software*, 83, 723-744.
- [265] Zhao, J. (1998). Applying Slicing Technique to Software Architectures. *4th IEEE International Conference on Engineering of Complex Computer Systems*, 87-98.
- [266] Zhao, J., Yang, H., Xiang, L., & Xu, B. (2002). Change Impact Analysis to Support Architectural Evolution. *Journal of Software Maintenance: Research and Practice*, 14(5), 317-333.

- [267] Zowghi, D., & Gervasi, V. (2003). On the interplay between consistency, completeness and correctness in requirements evolution. *Information and Software Technology*, 45, 993-1009.
- [268] Zowghi, D., & Offen, R. (1997). A logical framework for modeling and reasoning about the evolution of requirements. *RE 1997*, 247-257.