

Establishing and Maintaining Semantically Rich Traceability: A Metamodelling Approach

Nikolaos Matragkas

Department of Computer Science

University of York

A thesis submitted for the degree of

Doctor of Philosophy

March 2011

Abstract

This thesis addresses the problem of model-to-model traceability in Model Driven Engineering (MDE). A MDE process typically involves models expressed in different modelling languages that capture different views of the system under development. To enhance automation, consistency and coherency, establishing and maintaining semantically rich traceability links between models used throughout the software development lifecycle is of paramount importance.

This thesis deals with the various challenges associated with providing traceability support in the context of MDE by defining a domain-specific, model-based traceability approach, which supports the main traceability activities in a rigorous and semi-automatic manner. To evaluate the validity of the thesis proposition, a reference implementation has been provided. The results obtained from the application of the proposed approach to various case-studies and examples have confirmed the feasibility and benefits of such an approach.

Contents

Nomenclature	xi
1 Introduction	1
1.1 Motivation and Research Challenges	2
1.1.1 Software Traceability in MDE	2
1.2 Hypothesis and Objectives	4
1.2.1 Definition of Case-Specific Traceability Models	5
1.2.2 Identification of Trace Links	5
1.2.3 Maintenance of Traceability Models	6
1.2.4 Usage scenarios of Traceability in MDE	6
1.3 Research Results	6
1.4 Thesis Structure	7
2 Background	10
2.1 Defining Traceability	10
2.2 Benefits of Traceability	13
2.3 Traceability Modes	16
2.3.1 Backward and Forward Traceability	17
2.3.2 Pre and Post-Requirement Specification Traceability	17
2.3.3 Horizontal and Vertical Traceability	18
2.3.4 Implicit and Explicit Traceability	18
2.3.5 Functional and Non-Functional Traceability	19
2.4 Traceability Related Activities	19
2.4.1 Traceability Identification	20
2.4.1.1 Information Retrieval Approaches	23

2.4.1.2	Rule-Based Approaches	27
2.4.1.3	Miscellaneous Approaches	30
2.4.2	Traceability Representation	34
2.4.2.1	Specification of Traceability Information	34
2.4.2.2	Actualisation and Visualisation of Traceability Information	44
2.4.3	Traceability Maintenance	48
2.4.3.1	Event-Driven Traceability Maintenance	49
2.4.3.2	State-based Traceability Maintenance	52
2.4.4	Traceability Usage	52
2.4.4.1	Traceability for change management and impact analysis	53
2.4.4.2	Traceability for V&V activities	54
2.4.4.3	Traceability for Testing	55
2.4.4.4	Traceability for Understanding the System	56
2.4.4.5	Traceability for Reuse	56
2.4.4.6	Traceability for Software Project Management	57
2.5	Traceability in Practice	57
2.5.1	Traceability Tools	58
2.5.1.1	Tools with Traceability Support	58
2.5.1.2	Specialised Traceability Tools	59
2.5.1.3	Tool chains	60
2.5.2	Empirical Studies	61
2.5.3	Limitations	64
2.5.3.1	Natural Factors	64
2.5.3.2	Economic Factors	65
2.5.3.3	Social Factors	67
2.6	Concluding Remarks on Traceability	67
2.7	Model Driven Engineering	67
2.7.1	A Basic Theory of Models	68
2.7.2	Metamodeling	72
2.7.3	MDE: Concepts and Practices	73
2.8	Chapter Summary	77

3	Analysis of Software Traceability & Hypothesis	79
3.1	Introduction	79
3.2	Phenetics	81
3.3	Discussion	91
3.3.1	Analysis of the Data Matrix	91
3.3.2	Interpretation of the Dendrogram	94
3.4	Research Hypothesis	97
3.5	Research Scope	101
3.6	Research Methodology	101
3.6.1	Analysis	102
3.6.2	Design and Implementation	102
3.6.3	Testing	103
3.7	Chapter Summary	103
 4	 A Metamodelling Approach to Traceability	 104
4.1	Representation of traceability with TML	106
4.1.1	Strongly Typed Trace-links Conforming to a Case-Specific Meta- model	107
4.1.2	Correctness Constraints	108
4.1.3	Recurring Patterns in Case-Specific Traceability Metamodels . .	109
4.1.4	The Traceability Metamodelling Language	111
4.1.4.1	Abstract Syntax	112
4.1.4.2	Semantics	114
4.1.4.3	Concrete Syntax	123
4.1.4.4	Traceability Representation - Examples	124
4.2	Trace Link Recovery with TML	134
4.2.1	Summarising Current Practices to Traceability Identification . .	134
4.2.2	Requirements for Traceability Identification	138
4.2.3	Identification in TML	138
4.2.3.1	Internal Trace to Generic Trace	140
4.2.3.2	From a Generic Trace Model to a Traceability Model	146
4.2.4	Traceability Recovery with TML - Conclusions	149
4.2.5	Traceability Identification - Example	151

4.3	Maintenance of traceability with TML	156
4.3.1	Link Types	156
4.3.2	Model Change Types	156
4.3.3	Summarising Current Practices to Traceability Maintenance . .	157
4.3.4	Requirements for Traceability Maintenance	158
4.3.5	State-Based Traceability Maintenance in TML	159
4.3.6	Maintenance of Traceability with TML - Conclusions	163
4.3.7	Traceability Maintenance - Example	164
4.4	Traceability usage in the context of MDE	168
4.4.1	Transformation Validation with TML	169
4.4.2	Change Propagation with TML	173
4.5	Chapter Summary	176
5	Reference Implementation	177
5.1	Eclipse Platform	178
5.2	Eclipse Modeling Framework	178
5.3	Epsilon Framework	179
5.3.1	Epsilon Object Language	180
5.3.2	Epsilon Transformation Language	181
5.3.3	Epsilon Generation Language	181
5.3.4	Epsilon Validaton Language	181
5.3.5	Epsilon Comparison Language	181
5.3.6	Epsilon Wizard Language	182
5.4	Eclipse Views & Editors	182
5.5	Launch Configuration Interface	183
5.6	Availability	183
5.7	Chapter Summary	183
6	Evaluation	186
6.1	Means of Evaluation	187
6.1.1	Traceability scenarios	188
6.1.2	Peer review	188
6.1.3	Case study	189

6.1.3.1	Graphical Modelling Framework	190
6.1.3.2	EuGENia	191
6.1.3.3	The filesystem metamodel	193
6.1.3.4	Defining Traceability between Ecore and GMF Models	196
6.1.3.5	Establishing traceability	203
6.1.3.6	Using and maintaining traceability	204
6.2	Evaluation of the contributions	210
6.2.1	Classification of Traceability Approaches (Section 3.2)	210
6.2.2	Traceability Metamodeling Language (Section 4.1)	211
6.2.3	Traceability Identification with TML (Section 4.2)	212
6.2.4	Traceability Maintenance with TML (Section 4.3)	213
6.2.5	Traceability Usage (Section 4.4)	213
6.3	Evaluation of the Thesis Proposition	214
6.4	Shortcomings and Limitations	215
6.4.1	Lack of support for non-model artefacts	215
6.4.2	Lack of support for custom traceability information	215
6.5	Chapter Summary	216
7	Conclusions and Future Work	217
7.1	Review Findings	218
7.2	Proposed Solution	219
7.2.1	Traceability Metamodeling Language	219
7.2.2	Traceability Identification with TML	220
7.2.3	Traceability Maintenance with TML	220
7.2.4	Traceability Usage	221
7.3	Evaluation Results	221
7.4	Areas of Further Work	222
7.4.1	Support for non-model artefacts	222
7.4.2	Expand the concept of DSM2L	222
A	Characters Used in the Phenetic Analysis	223
B	Phenetic Analysis Data	227

C Phenetic Analysis Results	233
D TML Transformations	244
E EVL Constraints for the Filesystem Case-Study	250
References	279

List of Figures

2.1	Traceability Definitions	14
2.2	Traceability Process	21
2.3	Reactive Traceability (Costa & da Silva, 2007)	22
2.4	IR Models (Kuropka, 2004)	24
2.5	Traceability Metamodel according to (Ramesh & Jarke, 2001)	36
2.6	Requirements Interdependencies Classification (Dahlstedt & Persson, 2003)	39
2.7	Traceability Metamodel proposed by (Amar <i>et al.</i> , 2008)	41
2.8	Traceability Metamodel proposed by (Jouault, 2005)	42
2.9	AMW core metamodel by (Fabro <i>et al.</i> , 2005)	44
2.10	Traceability relating problem statements and solutions by (Arkley & Riddle, 2005)	55
2.11	TraceM conceptual framework (Sherba <i>et al.</i> , 2003)	62
2.12	A hierarchical structure for modeling in software engineering (Cowling, 2005)	70
2.13	Traceability Definitions	74
3.1	Dendrogram of Traceability Approaches	90
3.2	Basic relations of representation and conformance in MDE (adapted from (Jouault <i>et al.</i> , 2009))	99
3.3	Overview of the research methodology	102
4.1	Conceptual diagram of the TML approach	105
4.2	The ComponentClassTraceMetamodel Traceability Metamodel	110
4.3	Abstrax Syntax of TML	112

4.4	TML model in the Exeed Editor	124
4.5	Example of the Properties Pane	125
4.6	The <i>ComponentClassTraceMetamodel</i> specified using TML	126
4.7	The SD Metamodel of the i* Framework	129
4.8	The Sample OO Metamodel	129
4.9	The <i>IStar2OO</i> metamodel	131
4.10	Generic Trace Metamodel	140
4.11	Transformation-Chain	141
4.12	ETL launch configuration	148
4.13	Restaurant Booking System Class Model	151
4.14	ETL with TML run configuration	153
4.15	Restaurant Booking System Component Model	154
4.16	Snapshot of the generated traceability model	154
4.17	Class and Component Models for the Restaurant Example	165
4.18	Class-to-Component TML Model	166
4.19	Properties of the <i>getPackageContents ReconciliationExpression</i>	167
4.20	Faulty Class-2-Component ETL transformation - 1 _{st} case	172
5.1	The architecture of the Eclipse Platform	179
5.2	The architecture of the Epsilon framework (Kolovos <i>et al.</i> , 2007)	180
5.3	Example of the Exeed TML editor	183
5.4	Illustration of TML context menu	184
5.5	Illustration of the modified launch configuration of ETL	185
6.1	GMF Overview (Eclipse Foundation, 2011b)	190
6.2	EuGENia workflow (Kolovos <i>et al.</i> , 2010)	192
6.3	Graphical editor of the filesystem DSL (Kolovos <i>et al.</i> , 2010)	196
6.4	TML model for filesystem case study	197
6.5	Illustration of the EuGENia Ecore metamodel	200
6.6	Illustration of the filesystem traceability model	204
6.7	Illustration of the <i>RenameClass</i> menu	207

List of Tables

2.1	TEAP activities	43
2.2	Sample Traceability Matrix	45
3.1	Operational Taxonomic Units for the phenetic analysis	83
3.2	Attribute table	85
3.3	Similarity/Dissimilarity Coefficients Formulas	87
3.4	CCC for the Different Dendrograms	88
3.5	Link Semantics in Relation to the Automation of the Identification Activity	92
4.1	Traceability Identification Techniques	135
4.2	Decision table for overall matching result for a given link end	163
B.1	List of Character States	227
B.2	States Possessed by Each Character	232
C.1	Resemblance Matrix using the Dice Coefficient - Part I	234
C.2	Resemblance Matrix using the Dice Coefficient - Part II	235
C.3	Resemblance Matrix using the Hamming Coefficient - Part I	236
C.4	Resemblance Matrix using the Hamming Coefficient - Part II	237
C.5	Resemblance Matrix using the Jaccard Coefficient - Part I	238
C.6	Resemblance Matrix using the Jaccard Coefficient - Part II	239
C.7	Resemblance Matrix using the Kulczynski Coefficient - Part I	240
C.8	Resemblance Matrix using the Kulczynski Coefficient - Part II	241
C.9	Resemblance Matrix using the Sokal-Sneath Coefficient - Part I	242
C.10	Resemblance Matrix using the SokalSneath Coefficient - Part II	243

Author's Declaration

Except where stated, all the work contained in this thesis represents the original contribution of the author.

Parts of this work have been previously published by the author as journal, conference and workshop papers:

- Nicholas Drivalos, Richard F. Paige, Kiran Fernandes and Dimitrios S. Kolovos. *Towards Rigorously Defined Model-to-Model Traceability*, in Proc. 4th Workshop on Traceability, ECMDA'08, Berlin, Germany, June 200
- Nicholas Drivalos, Dimitrios S. Kolovos, Richard F. Paige, Kiran J. Fernandes. *Engineering a DSL for Software Traceability*, in Proc. 1st International Conference on Software Languages Engineering, SLE '08, Toulouse, France, Sept 2008
- Steffen Zschaler, Dimitrios S. Kolovos, Nikolaos Drivalos, Richard F. Paige and Awais Rashid. *Domain-Specific Metamodelling Languages for Software Language Engineering*, in Proc. 2nd International Conference on Software Language Engineering, Colorado, USA, October 2009
- Richard F. Paige, Nicholas Drivalos, Dimitrios S. Kolovos, Chris Power, Goran K. Olsen and Steffen Zschaler. *Rigorous Identification and Encoding of Trace-Links in Model-Driven Engineering*, Journal of Software and Systems Modelling, Springer, 2010 - accepted and to appear

- DOI: 10.1007/s10270-010-0158-8

- Louis M. Rose, Dimitrios S. Kolovos, Nicholas Drivalos, James R. Williams, Richard F. Paige, Fiona A.C. Polack, and Kiran J. Fernandes. *Concordance: An Efficient Framework for Managing Model Integrity*, in Proc. 6th European Conference on Modelling Foundations and Applications (ECMFA), June 2010, Paris, France
- Nicholas Drivalos-Matragkas, Dimitrios S. Kolovos, Richard F. Paige, Kiran J. Fernandes. *A state-based approach to traceability maintenance*, in Proc. of the 6th ECMFA Traceability Workshop, Paris, France, June 2010

Chapter 1

Introduction

“Software entities are more complex for their size than perhaps any other human construct because no two parts are alike (at least above the statement level). If they are, we make the two similar parts into a subroutine—open or closed. In this respect, software systems differ profoundly from computers, buildings, or automobiles, where repeated elements abound” (Brooks, 1987). It is this inherent complexity that makes software development a challenging activity. Various methodologies and techniques have been proposed and adopted over the years to reduce this complexity. The majority of those approaches is based on the principles, of abstraction, separation of concerns and problem decomposition (Dijkstra, 1976).

Model Driven Engineering (MDE) is a state of the art approach to software development that realises the aforementioned principles. It is primarily concerned with reducing the accidental complexity of software development through the use of technologies, which support the automatic transformation of problem-level abstractions to software implementations. The premise of the MDE paradigm is that models should be used to describe complex systems at different levels of abstraction and from a variety of viewpoints. These models should then be consecutively refined until they reach a state, where the final software system can be derived from them. Therefore, MDE promotes models to first class artefacts.

Following (Aizenbud-Reshef *et al.*, 2006a), each model in MDE can possibly have its own notation, representation, tools and users. Moreover, in an MDE process the developers, the tools, the artefacts and the processes are weakly integrated resulting to im-

explicit interconnections between them. consequently, maintaining consistency between the artefacts or propagating changes can be challenging.

This chapter highlights briefly the problems that motivated the work in this thesis. Following that, it outlines the research hypothesis and objectives and provides a summary of the results and the main contributions of this research work. Finally, it provides an overview of the organization of the thesis and a summary of the remaining chapters.

1.1 Motivation and Research Challenges

Models in MDE can take on different perspectives. Furthermore, (Kent, 2003) argues that different perspectives can possibly require modelling languages with different properties. As a consequence, during software development, it is possible that the system is modeled from different perspectives, which could be possibly expressed in different languages. However, since those models describe the same system (just from different viewpoints and at different levels of abstraction) they are implicitly related. Such implicit relationships can not be easily used to maintain consistency between the various development artefacts. Researchers have proposed the use of traceability relationships to articulate the aforementioned implicit interdependencies (e.g. (Aizenbud-Reshef *et al.*, 2006a; Paige *et al.*, 2008; Walderhaug *et al.*, 2008)).

Traceability can be considered as any relationship that exists between artefacts involved in software development life cycle (Aizenbud-Reshef *et al.*, 2006a). Such relationships include mappings, that are generated as the result of forward or backward transformations, links that are computed based on existing information or statistically inferred links.

1.1.1 Software Traceability in MDE

Researchers have associated different aspects of software traceability to a set of activities in an attempt to structure the field of traceability research (e.g. (Spanoudakis & Zisman, 2005; von Knethen & Paech, 2002)). Although those activities vary in the literature, four of them are common and are considered to be at the crux of traceability. These four activities are:

1.1 Motivation and Research Challenges

- *Representation*: in this activity the entities to be traced and the valid traceability relationship types between them are defined.
- *Identification*: in this activity previously unknown traceability relationships are identified.
- *Maintenance*: this is the activity of modifying existing traceability relationships in order to keep them in a valid state, while the entities, they refer to, change.
- *Usage*: this is the activity of using previously retrieved traceability information to support software development scenarios.

Various traceability approaches have been proposed in the literature in order to support the aforementioned activities. However, the majority of those approaches is not developed specifically for providing traceability support in the context of MDE. As a result, they do not fulfil the requirements of a traceability approach which targets MDE processes. Since models are the primary artefacts used in MDE, one of the requirements of a traceability approach is to be able to support traceability between models. This means that a traceability approach should consider both the structural information of models as well as their textual content in order to establish meaningful relationships. Currently though the majority of the traceability approaches gives an emphasis on the lexical content of artefacts ignoring their structural information.

Furthermore, in MDE various modelling languages are used to express models. An approach which focuses on MDE, should be extensible in order to facilitate support for heterogeneous modelling languages. Currently, many of the existing approaches focus on particular languages or on particular modelling artefacts, thus limiting their applicability to specific scenarios.

Finally, since models can possibly conform to different metamodels and have different semantics, traceability information between them can possibly have different semantics. Moreover, depending on the use of traceability different semantics might apply to traceability information between the same models. Therefore, in this context a traceability approach should be able to define the semantics of traceability information for the various traceability scenarios.

1.2 Hypothesis and Objectives

Based on the above discussion, the thesis proposition is stated as follows:

This thesis demonstrates that a domain specific model-based traceability approach can support and automate the process of rigorously managing the different types of heterogeneous traceability relationships between both derived and initial models in an MDE process.

The main characteristics highlighted in the above statement are the following:

1. **Model-based Approach:** In the spirit of MDE the proposed approach shall be applicable to model artefacts.
2. **Manage Traceability:** The proposed approach will provide support for all four traceability activities, i.e. representation, identification, maintenance and usage of traceability.
3. **Heterogeneous Traceability Relationships:** Using the proposed approach, an engineer will be able to manage traceability relationships between models expressed in heterogeneous notations.
4. **Rigorous:** The proposed approach will support the capture of case- specific or scenario-specific semantics.
5. **Automation:** The proposed approach will provide semi-automatic identification and maintenance of traceability information.
6. **Derived and Initial Models:** The proposed approach will be suitable for traceability between models, which are generated automatically by applying model operations on other models (derived models) as well as between models, which are created manually by engineers (initial models).

Motivated by the aforementioned limitations of the existing approaches, this research focuses on the following objectives:

- To propose an approach, which enables the construction of rigorous, well-defined, case-specific traceability models.

- To support and automate the activity of identifying traceability links between models.
- To support and automate the activity of maintaining traceability information between models.
- To propose novel usage scenarios of traceability information in MDE.

In the sequel, an overview of the research objectives is provided.

1.2.1 Definition of Case-Specific Traceability Models

It is widely accepted in the literature that the meaning of trace links should be precisely defined (e.g. (Aizenbud-Reshef *et al.*, 2006a; Walderhaug *et al.*, 2008)). A precise semantics for traceability will allow developers to capture more accurately the intended meaning of a specific traceability relationship, and will enable richer, tool-supported analysis and reasoning about traceability. In the spirit of MDE, a well-defined traceability model with case-specific semantics should conform to a case-specific traceability metamodel and a set of correctness constraints. Therefore, this thesis aims to propose an approach, which will facilitate the definition of traceability metamodels and of their accompanying correctness constraints in order to support the definition of traceability models with rich, case-specific semantics.

1.2.2 Identification of Trace Links

As mentioned in section 1.1.1, the computation of traceability information is one of the main traceability activities. Since manual computation is considered to be costly, the automation of this activity is of paramount importance. Automation however should not come at the expense of the quality of the computed traceability models. Those models should still possess important characteristics, such as precisely-defined and case-specific semantics. One of the objectives of this thesis is to define an approach, which will support the automatic computation of well-defined traceability models in the context of MDE.

1.2.3 Maintenance of Traceability Models

The activity of traceability maintenance deals with the upkeep of the integrity of computed trace links, while the entities, they refer to, continue to change and evolve. Traceability maintenance is considered to be of paramount importance, since it contributes to the avoidance of the degradation of traceability information. Maintaining traceability models manually can be a time consuming and error prone activity. Hence, this thesis aims to define an approach, with which the validity of trace links can be maintained in a semi-automatic manner.

1.2.4 Usage scenarios of Traceability in MDE

Many different traceability usage scenarios have been identified in the literature. These scenarios capture different ways in which traceability information can be used to support the development of software systems. The identified scenarios are usually generic and apply to software development in general. One of the objectives of this work is to identify usage scenarios, if any, which are MDE-specific. That is, scenarios which are encountered mainly in MDE processes.

1.3 Research Results

As a result of this thesis, a holistic approach to model-to-model traceability support for MDE processes has been proposed. This approach is considered to be holistic, since it supports all four of the main traceability activities; namely the definition of traceability information, the computation of traceability models, the maintenance of computed traceability models and finally the use of such models.

The thesis has also contributed a novel classification of the existing traceability approaches. This classification of traceability approaches has been developed using the technique of numerical taxonomy. It is considered to be unique and novel in the sense that to our knowledge none of the existing traceability classifications refers to, or applies, any concepts or techniques from the science of classification.

The thesis proposition has been validated by demonstrating that scenario specific traceability models with rich semantics, can be defined, computed, maintained and support MDE activities using the proposed approach. Finally, acceptance by the MDE

research community has provided confidence for the applicability and validity of the proposed approach.

1.4 Thesis Structure

This thesis contains seven chapters, structured as follows. In chapter 2 a detailed review of the related work is performed. More specifically, in section 2.1 various traceability definitions are considered and a working definition of traceability is provided. Section 2.2 presents the importance of traceability in the software development process, while in section 2.3 the main traceability modes are illustrated. The main section of this chapter is section 2.4. In this section the four traceability activities are analysed and the approaches related to each of the four activities are introduced. More particularly, in section 2.4.1 various techniques for the computation of traceability models are presented. These techniques include Information Retrieval (IR) methods, rule-based approaches, history analysis approaches and run-time monitoring methods. In the next section, that is section 2.4.2, the activity of traceability representation is briefly introduced and the most influential approaches related to this activity are presented. Section 2.4.3 summarises the literature related to traceability maintenance, while section 2.4.4 identifies and briefly describes the various traceability usage scenarios reported in the traceability literature. Following that, section 2.5 discusses the use of traceability in industrial settings. In this section, popular tools are presented, empirical studies related to traceability are introduced and finally limitations, which affect the adoption of traceability in the industry, are identified. Finally, the second part of chapter 2, that is section 2.7, consists of a brief discussion on the main concepts and practices related to MDE.

Chapter 3 summarises the findings of the review performed in chapter 2 and it provides a formal framework for understanding the different areas of traceability research. This is achieved by developing a classification of traceability approaches using *phenetics*, which is a classification approach borrowed from Biology. Based on the phenetic analysis, limitations of the contemporary traceability approaches are identified and the thesis proposition as well as the research objectives are outlined. More particularly, in section 3.2 the phenetic analysis is performed. The outcome of this analysis is an attribute matrix and a classification captured in a tree-like structure called *dendrogram*.

Section 3.3 provides a discussion on the findings of the phenetic analysis and on identified open issues in the domain of traceability. Based on this discussion, the research context and the thesis proposition are established in section 3.4. Finally, in section 3.6, the research methodology that is followed in order to evaluate the validity of the thesis proposition is presented.

Chapter 4 presents a novel approach to model-to-model traceability support for MDE processes. This approach focuses on how to define and implement semantically-rich trace links that can become amendable to automatic manipulation. More precisely, section 4.1 defines the *Traceability Metamodelling Language* (TML), which is a domain-specific metamodelling language for traceability. The abstract and the concrete syntax of TML are presented as well as its semantics. Section 4.2 presents how traceability models, which conform to scenario-specific traceability metamodels, can be computed in a semi-automatic manner, while section 4.3 illustrates how the validity of traceability models can be maintained, when the models they refer to change. This is achieved by using maintenance-specific metadata captured in TML models as well as traces generated from model management tasks. Finally, section 4.4 presents two novel usage scenarios of traceability in the context of MDE. More particularly, section 4.4.1 illustrates how TML models can be used to identify problematic sections of model transformation specifications. In section 4.4.2 TML models are used in conjunction with model transformation languages to propagate changes to affected models during model refactorings.

In chapter 5, a discussion on interesting aspects of the reference implementation is provided. Such aspects include the choice of the Eclipse platform and the Epsilon framework atop which the tool which supports the proposed approach is built. Furthermore, how the proposed approach is integrated with the Epsilon framework is presented.

Chapter 6 presents the evaluation of the proposed contributions against the thesis proposal. Section 6.1 describes the means of evaluation, which were employed. More particularly, in section 6.1.1 the use of examples and short case study for evaluating the various concepts of this thesis is illustrated. Section 6.1.2 evaluates the impact of this research work in terms of visibility and acceptance by the MDE community. In section 6.1.3, a case study, which was used to evaluate the proposed approach, is illustrated. In section 6.2 the contributions of this thesis are examined individually, while in section

6.3 an evaluation of the thesis proposition is provided. Finally, section 6.4 discusses the limitations and shortcomings of the proposed approach.

Chapter 7 concludes by summarizing the findings and contributions of this thesis. Furthermore, it discusses how the thesis proposition is supported by the thesis contributions. Finally, it provides directions to further work in the field of traceability for MDE processes.

Appendix A identifies and describes briefly the various characters used in the phenetic analysis conducted in chapter 3. Appendix B illustrates the different states, which can be possessed by each of the characters described in appendix A, while in appendix C the detailed results of the phenetic analysis are presented. Appendix D provides the complete transformations of TML. Finally, in appendix E the generated constraints of the case study, which is presented in chapter 6, are provided.

Chapter 2

Background

The aim of this chapter is to illustrate and to provide a critical review of the research relevant to the work presented in this thesis. The review is separated into two main sections. The first section discusses software traceability and outlines its importance and principles. It describes the main traceability activities, namely identification, representation, maintenance and usage, as well as the related literature. Finally, it presents briefly some of the most widely used traceability tools and it identifies some of the issues which have hindered the adoption of traceability practices in industry. The second section describes Model Driven Engineering (MDE), whose principles are used extensively in this work. This section starts with a brief discussion about models and modeling in general in software engineering. This is followed by a presentation of the basic metamodeling concepts and the use of metamodels in MDE. Finally, the chapter is concluded with a brief discussion of the main MDE principles and practices.

2.1 Defining Traceability

It is challenging to locate traceability research in the computer science and software engineering research map, since it overlaps with different domains of interest, including requirements engineering (RE), knowledge engineering and software project management. Consequently, the existing terminology and definitions are not entirely standardised, since they depend heavily on the domain in which they are used. A study of the existing body of literature suggests that the notion of traceability has its origins in the

field of RE. Many researchers have made their focus mainly on the relationship between a system's requirements specification and other artefacts of the software development process, or even more on the association of the requirements specification with various stakeholders (e.g. (Gotel & Finkelstein, 1994; Ramesh & Jarke, 2001)). Thus, the terms *requirements traceability* and *traceability* are very often used interchangeably. In recent years, the research community has started to delve into traceability without associating it strictly to requirements. That is to say, traceability today is understood as a more comprehensive concept encompassing the whole of the development process, without putting special emphasis on any phase of it (Aizenbud-Reshef *et al.*, 2006a; Winkler & von Pilgrim, 2009).

As is evident from the existing literature, traceability is a term that is open to considerable variation among definitions. Several definitions exist depending on how traceability is conceived and in which domain it is used. One of the earliest definitions of traceability has its origins in the work of (Gotel & Finkelstein, 1994):

“Requirements traceability refers to the ability to describe and follow the life of a requirement in both forward and backward direction, i.e from its origins through its development and specification, to its subsequent deployment and use, and through period of ongoing refinement and iteration in any of these phases.”

This definition has been formulated in the context of RE, hence it assumes that traceability applies only to the relation of requirements with other artefacts of the software development process. Notably, traceability is defined as a time dependency of a requirement and its associated artefacts, such as a requirement's implementation. However, this definition overlooks the fact that usually it is useful to interrelate artefacts that are created simultaneously, or artefacts that do not have a predecessor-successor relationship, such as two distinct requirements.

(Pinheiro, 2004) offers a similar definition to (Gotel & Finkelstein, 1994) by stating that “requirement traceability refers to 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.” Despite the fact that this definition is still heavily affected by the domain in which it is used, i.e. the RE domain, it is more

generic in the sense that it does not require the traceable artefacts to have a predecessor-successor relationship.

A domain-specific definition is provided by the Object Management Group (Object Management Group, 2011b). According to the OMG, “a **trace** records a link between a group of objects from the input models and a group of models from the output models. This link is associated with an element from the model transformation specification that relates the groups concerned”. This definition is provided in the context of Model Driven Engineering, and particularly in the context of model-to-model transformations. Therefore, such a definition is not sufficient to cover traceability in all the different varieties and forms, which are encountered in contemporary software engineering, thus limiting its usefulness in the MDE domain.

A more generic definition of traceability can be found in the IEEE’s Standard Glossary of Software Engineering Terminology (IEEE, 1990).

“The degree to which a relationship can be established between two or more products of the development process [...]”

In a similar way, (Aizenbud-Reshef *et al.*, 2006a) defines traceability “as any relationship that exists between artefacts involved in the software engineering life cycle”. These two definitions do not focus on any particular domain but rather define traceability with respect to products or artefacts of the development process. However, the terms product and artefact have a coarse-granular connotation, implying whole documents, models or code. As a result, parts of documents, model elements or small pieces of code are disregarded by those two definitions.

(Paige *et al.*, 2008) acknowledge the need to relate not only artefacts but also parts of artefacts, as long as these parts are unique and identifiable. Hence, they identify traceability as “the ability to chronologically interrelate uniquely identifiable entities in a way that matters”. Nonetheless, they impose a time constraint over what can be traced, since they imply that traceable entities can be only chronologically interrelated.

Finally, (Spanoudakis & Zisman, 2005) go one step further and they define traceability not only with respect to artefacts produced during the development process, but also with respect to stakeholders and artefact rationale.

“[Software traceability] is the ability to relate artefacts created during the development of a software system to describe the system from different perspectives and levels of abstraction with each other, the stakeholders that have contributed to the creation of the artefacts and the rationale that explains the form of the artefacts.”

Although this definition adds into the picture, stakeholders and rationale, it considers only whole artefacts and not artefact parts, thus it suffers from the same limitations as the definitions mentioned above.

The definitions described so far focus on particular attributes of traceability (e.g. time dependency). However, there are no definitions that provide a generic and inclusive framework, encompassing all attributes. The scope of this thesis requires a definition of traceability able to cover all potential cases. This inevitably results in a broader spectrum of attributes included (in the definition) capable of capturing all traceability cases (between models) identified during this work. Hence, for the purposes of this work, we will provide our own working definition, which is the following:

“Software traceability is the ability to interrelate uniquely identifiable entities, the stakeholders that have contributed to the creation of those entities and the rationale that explains their form in a way that matters.”

Figure 2.1 below illustrates one possible gamut, drawing attention to the way in which traceability is conceived by the research community from the more specific definitions to the more generic ones.

2.2 Benefits of Traceability

In the previous section, several definitions of traceability are provided. As can be seen, it is a very generic concept and easy to understand. What might not be obvious from the aforementioned definitions is why traceability is needed. In this section, the importance of traceability in the software development process will be illustrated.

The benefits of traceability are well documented in the literature. (Ramesh *et al.*, 1993) groups the various benefits of traceability according to the needs of the stakeholders involved in the development process. His results are based on a multitude of sources such as literature surveys, focus groups, interviews and empirical studies.

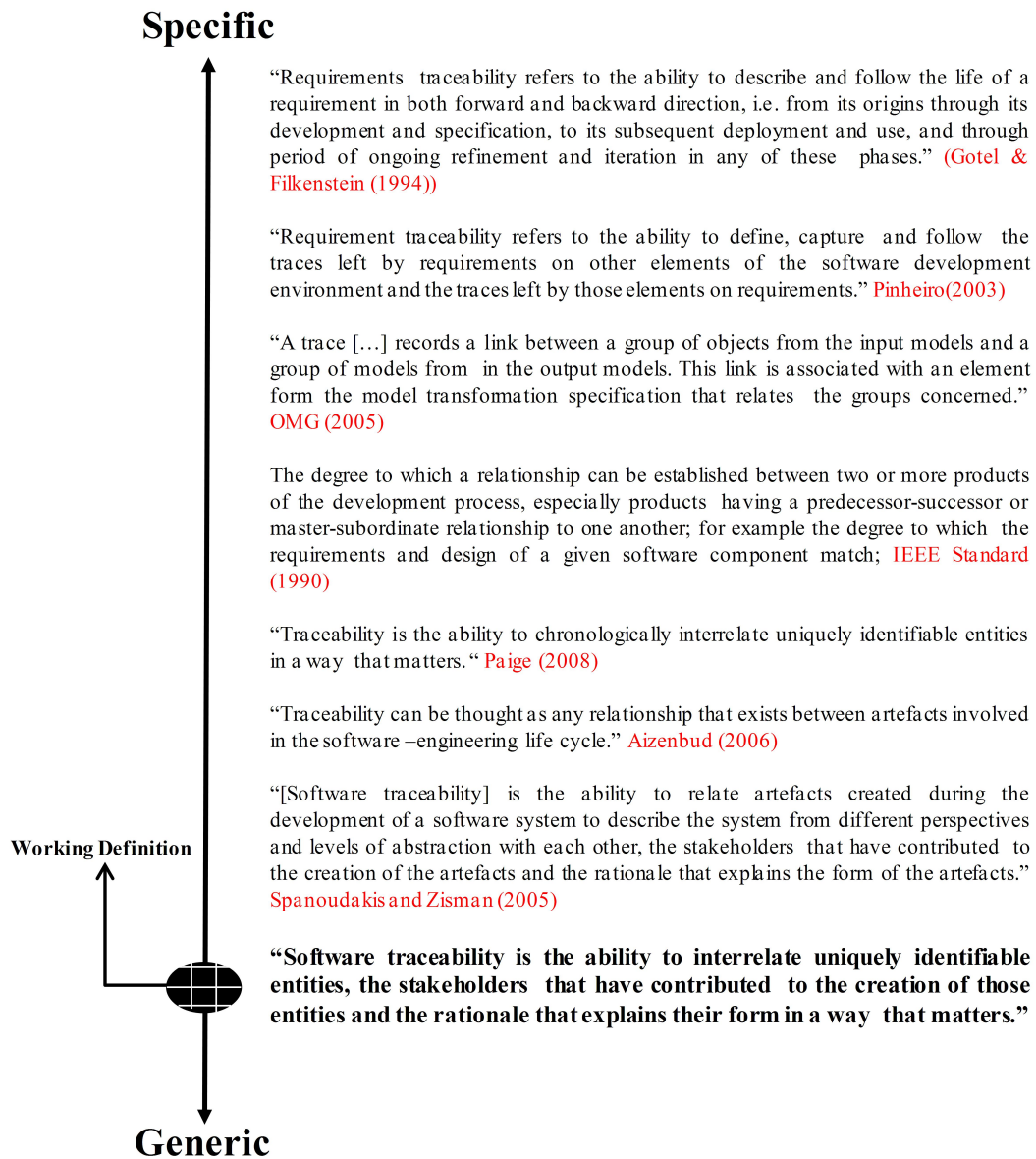


Figure 2.1: Traceability Definitions

From the project management point of view, traceability can be used to track project status. By using a tracking information system (i.e. traceability) the impact of change can be estimated. Moreover, conflicts between requirements can be discovered early and requirements not satisfied yet by an implementation can be identified. As a result, the work required to realize this requirements can be estimated. Finally, future systems could potentially have reduced development time and effort, since past implementation decisions can be tracked down and reused.

From the customer point of view, (Ramesh *et al.*, 1993) identifies several benefits related to traceability. If traceability information is captured and maintained, then the requirements that satisfy parts of the implementation can be identified. Additionally, tests which must be performed to ascertain the presence of a requirement can be determined. As a consequence, the quality of the product with respect to the user requirements can be evaluated. Acceptance testing can be associated directly to the user requirements being tested for. Finally, user requirements are linked to design decisions so that developers can keep their focus on those requirements they are trying to satisfy.

Traceability should ideally link the results of design, the rationale of the various design decisions, alternatives considered and the assumptions made in a decision. If these are recorded, a designer can benefit from traceability in several ways. First, she can verify more easily that a design satisfies the requirements. Furthermore, the designer can estimate the impact of a change in requirements on the design or the impact of a change in available implementation technology on the design assumptions. Lastly, design components can be reused in various projects, since the assumptions under which those components will work are recorded (Ramesh & Edwards, 1993).

The traceability benefits realised by a system maintainer have to do mainly with impact analysis. First, a maintainer can estimate the impact of a change in one requirement to others. Additionally, the impact of a change in a requirement on the implementation can be estimated. Finally, permissibility of changes in implementation can be estimated with respect to unchanged requirements.

Due to the aforementioned benefits, traceability is closely associated with software and system quality. There is a number of regulations, which dictate the use of traceability. Some of them are the following (Albinet *et al.*, 2007):

- *IEC 880* for nuclear systems

- *CENELEC EN 50126, EN 50128 and EN 50129* for railway systems
- *DO-178* and *DO-254* for aeronautic systems

Additionally, many quality standards recommend traceability as a desired feature of a system. These standards include the following:

- *IEEE Std. 1219*
- *ISO 9000*
- *ISO 15504*
- *CMMI*

Traceability can be considered a quality attribute since it helps developers ensure that they are building the *right* system in the *right* way (Behrens, 2007). First, traceability can be used to ensure that a software product has all the characteristics requested by the stakeholders and to ensure that the product does not have characteristics which are not requested by the stakeholders. In software engineering, this is called *software validation*, i.e. the process of checking that the product design satisfies or fits the intended usage. Moreover, traceability can be used to ensure that the software built conforms to its specification. Traces can be used to link various artefacts and their related tests. This use of traceability is related to the verification of software.

2.3 Traceability Modes

There are several ways in which tracing can be performed. With reference to time, an entity can be traced in a **forward** or **backward** direction. With regard to the evolution of entities, they may be traced to aspects occurring before or after their inclusion in formal documents of the development life-cycle such as the requirements specification. Finally with regard to the type of entities involved, we may have implicit or explicit traceability.

2.3.1 Backward and Forward Traceability

The direction of tracing can be an indication of time dependency or causality. Backward and forward traceability have been defined in the context of RE in the following way (Wieringa, 1995):

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

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

A requirement is traced forward, for example when the requirement is changed and the impact of this change needs to be investigated. A requirement is traced backward, for example, when there is a requirement change and we need to understand it, investigating the information used to elicit the changed requirement.

The ANSI/IEEE Std 830-1984, namely the IEEE Guide to Software Requirements Specifications, defines forward and backward traceability in a more generic fashion. Backward traceability refers to the ability to follow the traceability relationship from a specific artefact back to its source from which it has been derived. On the other hand, forward traceability is ability to follow a traceability relationship to an artefact which spawned from a source artefact.

2.3.2 Pre and Post-Requirement Specification Traceability

Another distinction of traceability is between pre and post-requirements specification (RS) traceability. This distinction is encountered mainly in the RE literature and it refers to the relationships a requirement has prior or after its inclusion in the RS document (Wieringa, 1995). More precisely, the tracing of a requirement can be done either for the purpose of getting information related to the process of elicitation of the requirement prior to its inclusion in the RS document or for the purpose of getting information about the requirement's use after its inclusion in it (Gotel & Finkelstein, 1994). This distinction can be very useful since the software development process is driven by the RS document. Therefore, if a requirement changes developers might want to find out what is the source of this change. Additionally, when a requirement changes, developers

need to find its interdependencies with other artefacts such as design modules, which can be affected by this change. (Gotel & Finkelstein, 1994) claim that pre-RS traceability is more challenging than post-RS traceability due to the informal nature of activities which take place during the requirements elicitation.

2.3.3 Horizontal and Vertical Traceability

The distinction between horizontal and vertical traceability is mentioned in the traceability literature as well. However, there is no common agreement as to what exactly is meant by those two terms. (Lindvall & Sandahl, 1996) define horizontal traceability as the relationships between different models and vertical traceability as the relationships between elements of the same model. (Boldyreff *et al.*, 2002) provide a different definition: the term horizontal traceability is defined as the relationships between artefacts developed in one stage of the development lifecycle, while the term vertical traceability is defined as the relationships between artefacts developed in different stages of the software development lifecycle.

2.3.4 Implicit and Explicit Traceability

Artefacts of the software development process can be traced explicitly or implicitly. Explicit traceability requires the engineer to manually establish a relationship between two artefacts. Implicit traceability results from an inherent relationship between the traceable items (Paige *et al.*, 2008). In most cases, traceability is handled explicitly.

(Aizenbud-Reshef *et al.*, 2006a) use the terms *imposed* and *inferred* traceability instead. An imposed traceability relationship is a relationship between entities that exists by volition of the relationship creator and it cannot be invalid until its creator determines that it is invalid. For example a *satisfies* relationship can be created between a requirement and an element of a design model. When the requirement changes it is not clear if the relationship is still valid or not. On the other hand an inferred relationship is one that exists because the related entities satisfy a rule that describes the relationship. An inferred relationship can not be invalid since if the rule is not satisfied the relationship does not exist. An example of an inferred relationship is the relationship which is a result of a model transformation.

(Maeder & Riebisch, 2007) provide a different interpretation of the two terms. Explicit traceability is described as any explicitly expressed relationship between entities, while implicit traceability is considered to be any interrelationship between entities which is implied but not explicitly represented. For example, a link between two elements, which belong to different models and have the same name, is considered to be an implicit link.

2.3.5 Functional and Non-Functional Traceability

Following (Pinheiro, 2004), traceability can be divided into *functional* and *non-functional*. Functional trace links are those links related to well established mappings between entities. This kind of trace is related to the functional aspects of software development, and more precisely those aspects that are described in terms of transformations on well-defined states of the artefacts of the software development process. Functional trace links are the result of syntactic and semantic interrelationships between the entities involved in the development process. The transformation from one artefact to the other does not have to be automatic, but it has to adhere to unambiguous and well-defined procedures. According to (Lindvall & Sandahl, 1996), this is called *seamless* transformation. An example of a seamless transformation is the transformation of an audio recording of a meeting to a transcript of the meetings minutes.

Conversely, non-functional traceability is related to the informal aspects of software development such as intentions, purpose, goals, responsibilities and other intangible concepts. (Pinheiro, 2004) classified the areas covered by non-functional traceability into four groups, namely *reason*, *context*, *decision* and *technical*.

2.4 Traceability Related Activities

Researchers have associated different aspects of traceability to a set of activities in an attempt to structure the field of traceability research (Pinheiro, 2004; Spanoudakis & Zisman, 2005; von Knethen & Paech, 2002). Although those activities vary in the literature, four of them are common and are considered to be at the crux of any traceability approach. The aforesaid activities are the following:

- Identification

- Representation
- Maintenance
- Utilisation

Figure 2.2 illustrates a simple traceability workflow. Initially, all possible links between two artefacts are identified. Then, these links are represented in an appropriate notation and stored. If a change is incurred by the referred entities and this change affects the recorded trace links then the links must be updated. Finally, when the use of traceability information is required during the software development process, the stored trace links are accessed and used. Although in the workflow the various traceability activities seem to follow a waterfall process model, this is not the case. How and when these activities take place depends heavily on the overall software development process followed in a project. In the early days of software engineering, the waterfall model was the predominant process used in software development, and thus traceability activities followed a linear process. Today iterative and agile processes are gaining in popularity, so traceability approaches are adjusting and are becoming more lightweight (e.g. (Lee *et al.*, 2003)). By and large, traceability activities are usually interleaved with one another as well as with the other activities of software development in a way which depends on the tracing needs of a project. At present, there is no standard nor a set of guidelines to practitioners on how to use the various traceability approaches within a standardize software development process.

In the following sections, the four identified activities are discussed in more detail, including related work.

2.4.1 Traceability Identification

Traceability identification is the activity of discovering previously unknown trace links and recording them in an appropriate format. The terms *trace recording* (Winkler & von Pilgrim, 2009), *trace generation* (Spanoudakis & Zisman, 2005), *trace creation* (Aizenbud-Reshef *et al.*, 2006a) and *trace production* (Pinheiro, 2004) are also used. The need for providing support for trace link identification has been widely reported in the traceability literature (Antoniol *et al.*, 2002; Egyed & Grunbacher, 2002; Marcus & Maletic, 2003; Murphy *et al.*, 1995; Zisman *et al.*, 2003). However, the majority

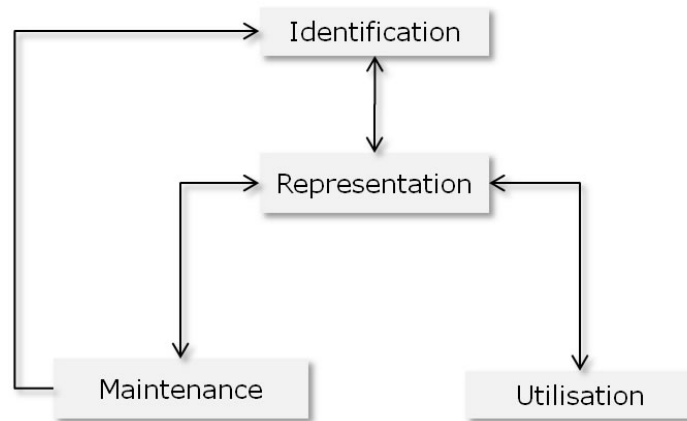


Figure 2.2: Traceability Process

of contemporary tools offer only limited support for traceability identification as they require the users to create trace links manually. The manual creation of trace links is expected in numerous approaches including (Bayerand & Widen, 2001; Dick, 2005; IBM, 2010; Pinheiro & Goguen, 1996; Pohl, 1996a). However, manual creation of trace links is considered as difficult, error-prone and time consuming. As a result, traceability is rarely established manually unless there is a regulatory reason to do so (Spanoudakis & Zisman, 2005).

Following (Costa & da Silva, 2007), there are two viewpoints associated with the identification and creation of trace links. First, artefacts can be considered as existent and traces will instantiate and describe some implicit or explicit semantic relation or dependency between them. The focus of traceability in this case is on creating and maintaining the relationships between **existing** artefacts. The identification and generation of trace links can be done manually or automatically by comparing the relevant artefacts and by using a predefined set of rules. This viewpoint is called the *dependency viewpoint*. On the other hand, an artefact can be generated by another artefact, using some type of transformation (manual or automatic). In this case, the trace links record relationships between the generated artefact and the one which was used for the generation. The identification and creation of the trace links is a byproduct of the transformation and in most cases can be done automatically. This viewpoint is the *generative viewpoint*. (Costa & da Silva, 2007) argue that these two approaches are not only valid but necessary in a development process and a combination of both should be used in

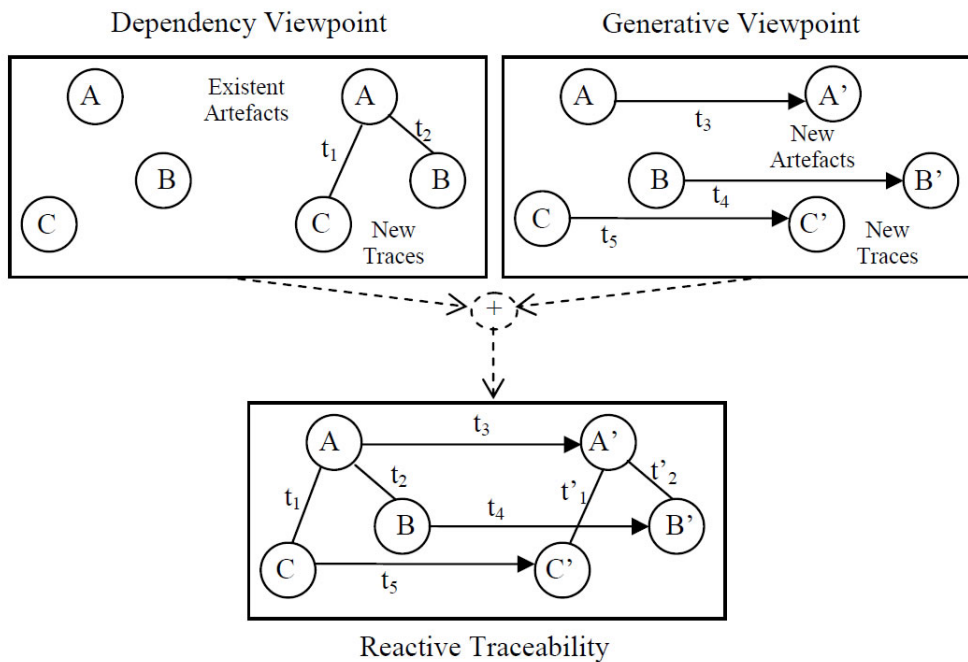


Figure 2.3: Reactive Traceability (Costa & da Silva, 2007)

every traceability approach to identification and generation of trace links. The combination of the two aforementioned approaches is called *reactive traceability* (Figure 2.3).

Similar to the distinction provided by (Costa & da Silva, 2007), (Winkler & von Pilgrim, 2009) argue that the identification and recording of traceability information can be performed either *on-line* or *off-line*. In the former case traces are identified and recorder automatically by tools as by-products of the development activity. In the latter case, traces are identified and created automatically or manually after the actual development activity has finished. The time elapsed between the completion of the development activity and the recording of traces affects the precision and the quality of the traces (Cleland-Huang *et al.*, 2003). The longer the time elapsed, the more imprecise the trace links are. In the following, a critical discussion of the main approaches to traceability identification is provided.

2.4.1.1 Information Retrieval Approaches

Information retrieval (IR) is the science of searching for documents, for information within documents, and for metadata about documents (Doyle & Becker, 1975). The standard problem of IR is the following: Given a collection of documents (*corpus*) and a query, determine those documents that are relevant to the query. IR works in two phases. In the first phase the system analyzes and indexes the document collection resulting into a representation for each item in the collection. This representation consists of all the keywords contained in each document. In the second phase, the system analyzes incoming queries and it constructs a representation of them. Finally it uses a matching algorithm to determine which document representations match the query representation.

IR techniques are used for trace link recovery on the assumption that two related documents will share the same vocabulary since developers normally choose names for the various entities from the application-domain knowledge. The corpus in the case of traceability consists of the various traceable artefacts.

The use of an IR method for the identification of trace links requires text intensive entities. Such methods do not rely on structural properties of the input and they are usually employed for off-line tracing.

For an IR method to be efficient, the documents are typically transformed into a suitable representation according to an appropriate mathematical model. Figure 2.4 illustrates the relationship of some common models, which are categorised according to their mathematical basis and the properties of the model. The *set-theoretic* models represent documents as sets of words or phrases. Similarities are usually derived from set-theoretic operations on those sets. On the other hand, *algebraic models* represent documents and queries as vectors, matrices, or tuples. The similarity of the query vector and document vector is represented as a scalar value. A popular algebraic model is the *Vector Space* model (VSM), which treats the weight of keywords in documents as vectors and it computes the similarity between documents as the *cosine* of the angle between the two vectors. A common criticism of VSM is that it does not take into consideration relationships among the various keywords. For example, having the keyword “automobile” in one document and the keyword “car” in another does not contribute to the similarity measure between those two documents. To overcome the synonymy and

2.4 Traceability Related Activities

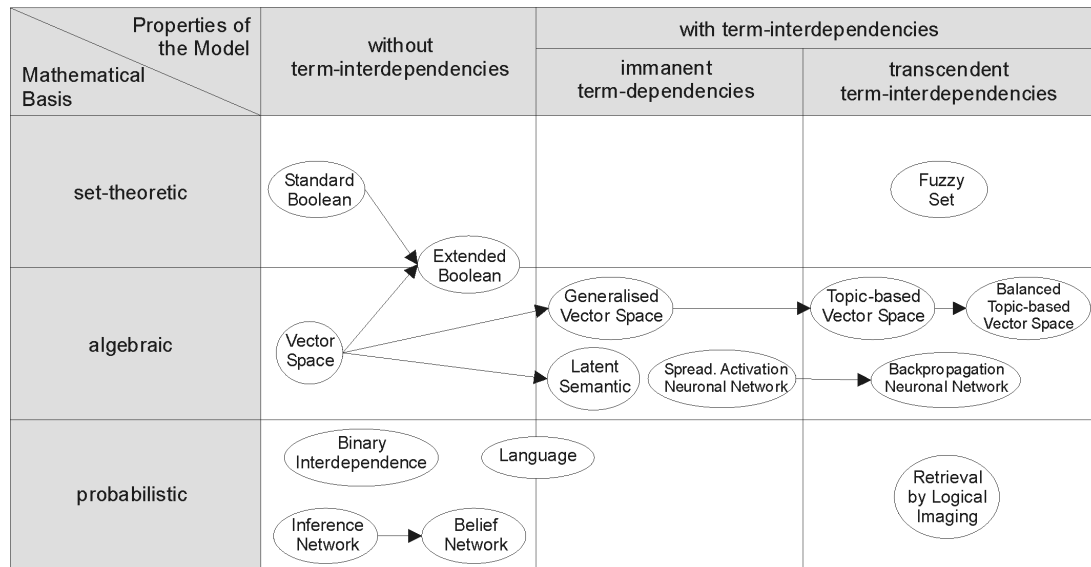


Figure 2.4: IR Models (Kuroпка, 2004)

polysemy limitations of VSM, (Deerwester *et al.*, 1990) developed the Latent Semantic Indexing (LSI) method. LSI uses a mathematical technique called Singular Value Decomposition (SVD) to identify patterns in the relationships between the keywords contained in a document. A key feature of LSI is its ability to extract the conceptual content of a body of text by establishing associations between those terms that occur in similar contexts. The last category of models used for document representation in IR consists of probabilistic models (PM). These treat the process of document retrieval as a probabilistic inference. The similarities between documents are computed as probabilities that a document is relevant for a given query.

The performance of the various IR techniques (and not only) is usually evaluated using two measures, *precision* and *recall*. *Precision* is defined as the fraction of retrieved documents which are relevant, while *recall* is defined as the fraction of relevant documents retrieved.

$$Precision = \frac{|\{relevantDocuments\} \cap \{retrievedDocuments\}|}{|\{retrievedDocuments\}|}$$

$$Recall = \frac{|\{relevantDocuments\} \cap \{retrievedDocuments\}|}{|\{relevantDocuments\}|}$$

A perfect *precision* score of 1.0 means that every result retrieved by a search was relevant (but says nothing about whether all relevant documents were retrieved) whereas a

perfect *recall* score of 1.0 means that all relevant documents were retrieved by the search (but says nothing about how many irrelevant documents were also retrieved). Usually, there is an inverse relationship between *precision* and *recall*, where it is possible to increase one at the cost of reducing the other. For example, when IR methods are used to retrieve trace links, the IR approach can often increase its *recall* by retrieving more trace links, at the cost of increasing number of irrelevant links retrieved, i.e. decreasing *precision*.

There is a wide variety of trace link recovery approaches based on IR. A brief discussion of the most influential approaches is presented in the remainder of this section.

(Hayes *et al.*, 2003) use a vector space IR method to automate the generation of trace links between natural language documents. In an attempt to reduce the number of missed or irrelevant trace links generated, they extend their initial VSM technique with the use of key-phrase lists and thesauruses. Their work has demonstrated that the use of a thesaurus can improve the recall of their approach (i.e. number of missed trace links) but it decreases the precision, i.e more irrelevant links are generated. Additionally, they have demonstrated that the use of thesauruses provides better results in terms of precision and recall than the results provided from the use of key-phrase lists. Based on their results, they have built the RETRO (REquirements TRacing On-target) tool (Hayes *et al.*, 2004). One of the features of RETRO is the improvement of the generated trace links based on user input. Finally, they have attempted to improve their results using and LSI model (Hayes *et al.*, 2006). However, the VSM based technique with the thesaurus extension outperformed the LSI-based one.

(Antoniol *et al.*, 2002) examine the use of IR techniques for trace link generation between requirements written in natural language and a source code component. In their approach, a list of identifiers and comments is extracted from the source code and this list is used to construct the corpus for the implementation. Queries are matched to the various documents (i.e. requirement documents and source code) using both PMs and VSMs. The results obtained using both models are quite similar.

In an attempt to improve the precision and recall problems of the existing IR retrieval techniques, (Cleland-Huang *et al.*, 2005) introduce three strategies for incorporating supporting information into a probabilistic retrieval algorithm. The strategies include hierarchical modeling, logical clustering of artifacts, and semiautomated pruning of the

probabilistic network. They investigate how the proposed strategies affect the performance of the probabilistic retrieval algorithm and they conclude that their strategies ameliorate the algorithm only when they are applied to a subset of the dataset, where the confidence for acceptance and rejection is low. (Zou *et al.*, 2006) improve the results achieved by these strategies by adding a new one, which utilises key phrases instead of key words. The outcome of this research effort is implemented in a tool called Poirot (Lin *et al.*, 2006). Poirot uses a probabilistic network model to generate trace links between requirements, design elements, code and other artefacts stored in distributed third party case tools such as DOORS (IBM, 2010) and source code repositories. The tool is designed with extensibility in mind, so that additional artefact types and third party tools can be added.

Another approach which automates the generation of trace links between requirement documents and source code is the one proposed by (Maletic *et al.*, 2003). Since they use an LSI model, their approach takes into account synonym terms. A trace link between two documents is established when the semantic similarity of those documents is greater than a threshold. In (State & Maletic, 2003), a comparison between this LSI-based approach and the VSM/PM-based approach of (Antoniol *et al.*, 2002) takes place. According to this comparison, LSI outperforms VSM and PM in terms of precision and recall.

More researchers have experimented with LSI-based models. (Lucia *et al.*, 2007) argues that using IR methods based on LSI models is not the optimal method for recovering trace links, since the number of false positives grows up too rapidly when the similarity of artefact pairs decreases below an *optimal threshold*. Moreover, this similarity threshold changes depending on the type of the artefacts and projects. As part of their work, they have developed Re-Trace (Lucia *et al.*, 2008), a trace link recovery tool in ADAMS (ADvanced Artifact Management System), which is an artifact-based process support system for the management of human resources, projects, and software artifacts (Lucia *et al.*, 2004). The main novelty proposed by (Lucia *et al.*, 2007) is the pre-categorisation of the analysed artefacts and the truncation of the list of possible candidates dynamically based on user feedback. This approach is shown to improve the performance of their algorithm.

(Lormans & van Deursen, 2005) and (Lormans & van Deursen, 2006) investigate what are the advantages of using LSI to track and trace requirements and what kind

of trace links can be recovered using LSI. Furthermore they offer an analysis of why correct trace links are missed and false links are retrieved. One of their interesting experimental findings is that tracing requirements in test cases is more accurate than tracing requirements in design.

Finally, (Natt och Dag *et al.*, 2002) and (Natt och Dag *et al.*, 2005) investigate how to apply IR techniques for the automatic detection of requirement interdependencies and duplication. They claim automated similarity analysis on a syntactic level using IR techniques may be effective in pinpointing true requirement duplicates and interdependencies. However, they conclude that their technique can not replace human judgment.

The aforementioned trace recovery methods are considered to be *after-the-fact* methods. That is, lists of possible candidates are returned by the algorithms and each trace link has to be reviewed and accepted or rejected by the user. Additionally, the approaches which rely on the use of IR methods for the trace link recovery, work on the assumption that the artefacts between which tracing occurs share the same terminology. In cases where the artefacts are not from the same domain, these approaches underperform. Another shortcoming of the approaches discussed in this section is the fact that they are not able to capture trace link semantics. That is, using IR techniques for trace recovery is useful only in finding a similarity between two artefacts. However, such techniques are not able to inform the user how the traceable artefacts are related.

2.4.1.2 Rule-Based Approaches

Another research direction for the recovery of trace links is the use of dedicated trace link recovery rules. These rules usually express heuristics which predict how different entities are linked, as well as the rationale of their relationship. Rule-based approaches can be applied both to models and text-based artefacts. This contrasts with the IR methods described in the previous section, which apply mostly to text-based artefacts such as requirements expressed in natural language or code.

One of the most basic rule-based approaches to the identification of trace link candidates is to search models or texts for occurrences of common keywords. An underlying assumption of this approach is that developers normally choose names for the various entities in a consistently. (Spanoudakis *et al.*, 2004) propose the use of XML-based rules for the generation of traceability between requirements statements,

use cases and analysis models. The artefacts to be traced are represented in XML and the generated trace links are represented as hyperlinks expressed in XLink (W3C, 2001), which is an XML markup language that provides methods for creating internal and external links within XML documents, and associating metadata with those links. Two different types of traceability rules are used in this approach. More specifically, *requirement-to-object-model* rules are used to trace the requirements and use cases to an analysis object model, and *inter-requirements traceability* rules are used to trace requirements and use cases to each other. The approach has been evaluated in numerous case studies, and the results achieved were promising, since the recall and precision measures achieved range between 50% and 95%. One shortcoming of this approach is the fact that the traceability engineer has to specify manually a complete set of traceability rules. In an attempt to address this shortcoming, (Spanoudakis *et al.*, 2003) developed a machine learning algorithm that produces traceability rules. The generation of such rules is informed by examples of traceability relations which are provided by the user and is based on a generalisation of other existing traceability rules.

In XTraQue (Jirapanthong & Zisman, 2007), the basic algorithm of (Spanoudakis *et al.*, 2004) is modified so that it can be applied to the automatic recovery of traceability relations between elements of documents generated in feature-based object-oriented methodologies. The traceability rules used in this work are classified into two groups, namely (a) direct rules, which support the creation of traceability relations that do not depend on the existence of other relations, and (b) indirect rules, which require the existence of previously generated relations. The documents are represented in XML and the rules are expressed in an extension of XQuery (W3C, 2007). This approach can generate nine types of traceability relations including satisfiability, dependency, overlaps, evolution, implements, refinement, containment, similar, and different relations between feature-based and object-oriented documents created during the development of product line systems such as process models, use cases, class diagrams, sequence diagrams, etc. The rules used in this approach take into consideration the semantics of the documents under consideration, the various types of traceability relations in the product line domain, grammatical roles of the words in the textual parts of the documents, synonyms and distance of words being compared. The work has been evaluated in terms of precision and recall in five different scenarios. The results of these experiments have shown an average precision of 85.3% and an average recall of 83.3%.

Machine learning techniques are used as well by (Grechanik *et al.*, 2007), whose approach is applied to the identification of trace links between use-cases and Java source code. The proposed solution is called LEarning and ANALyzing Requirements Traceability (LeanArt). It combines program analysis, run-time monitoring, and machine learning to automatically propagate a small set of initial trace links between program variables and elements of use case diagrams to additional unlinked program entities thereby recovering new trace links. The core idea of LeanArt is that after programmers initially link a few program entities to elements of the use cases, the system will collect adequate information from these links to recover trace links for the rest of the program automatically. This is achieved by monitoring the program execution traces and inferring the various reads and writes from and to the variables. Using this information in combination with the manually defined trace links, the learning algorithm can relate program entities with use case elements. LeanArt approach was evaluated on open-source and commercial applications written in Java. The obtained results show that after users link approximately 6% of the program entities to elements from use-case diagrams, LeanArt correctly recovers 87% of trace links in the best case, 64% on average, and 34% in the worst case. The variability of the experiment results is attributed to the fact that the LeanArt approach depends heavily on the consistent naming of the use-case elements and the program variables.

In TOOR (Traceability of Object Oriented Requirements) (Pinheiro & Goguen, 1996), requirements can be traced to design documents, specifications, code and other artefacts through user-defined relations. This allows developers to distinguish among different links between the same objects. Moreover, using mathematical properties such as transitivity, TOOR can relate entities which are not directly related. The tool is used in three phases. In the definition phase, classes of objects to be traced and of relations among them are defined in a project specification. In the registration phase, objects and relations are registered as the project evolves, using TOOR templates. Finally, in the extraction phase, traces are computed and presented.

(Cleland-Huang *et al.*, 2002b) and (Cleland-Huang & Schmelzer, 2003) use user input to recover trace links between requirements and performance models and between non-functional requirements and design code artefacts. Fine-grained trace links are dynamically generated during system maintenance and refinement based on user-defined links, which are specified during the inception, the elaboration and the construction of

the system. This approach supports trace recovery based on invariant rules of design patterns which are used to identify critical components of classes.

Most of the aforementioned rule-based approaches for the recovery of trace links perform satisfactorily in situations where terminology is used consistently in the different traceable artefacts. However, these approaches require substantial effort for setting up the basis for traceability, i.e. for specifying a complete set of traceability rules. Additionally, the flexibility of the rule-based approaches is limited in the sense that in many cases it is not easy to change the initial traceability infrastructure such as traceability rules, trace link types, etc, if the need arises.

2.4.1.3 Miscellaneous Approaches

In order to overcome the limitations associated with the IR and rule-based methods of trace link recovery, some alternative approaches have been proposed. In this section, we will briefly present the most influential of those approaches.

A strand of research on this area examines how the semi-automatic generation of traces can take place and what is the most efficient method for the manual definition of the various trace links. In this group of approaches, the manual declaration of trace links is assisted by the use of visualisation techniques. Usually, the various documents to be related are displayed in a tool and the users can select the elements to be related in an easier way. A representative approach of this philosophy is the approach created by (Alexander, 2003), who extends the requirements management tool DOORS (IBM, 2010), to allow the recovery and organisation of trace links between requirements and use cases.

Although, this approach provides the users with advanced support for visualising and navigating the traceable artefacts, the effort required to manually establish the trace links is still high, especially for large or complex artefacts. Moreover, no support is provided for defining the semantics of the trace links in an enforceable way. As a result, the correctness of the established trace links depends on the understanding of the semantics of the links by the users

Another research direction involves the analysis of the change history of the traceable artefacts or the analysis of their run-time behavior. (Ying *et al.*, 2004) and (Zimmermann *et al.*, 2004) have developed an approach that applies data mining techniques

to determine change patterns among sets of files that were changed together frequently in the past. The underlying assumption of their approaches is that the change patterns can be used to recommend potentially relevant source code to a developer performing a modification task. However, no guidance is provided for the type of the relationship of two related entities. (Wenzel *et al.*, 2007) describe an approach for deriving trace links between single model elements by comparing the version history of the models to which the model elements belong. This approach is limited only to similar elements in different versions of a model.

(Egyed & Grunbacher, 2005) use program execution traces to recover relations between source code, requirements and test-cases. In this approach, test-cases are manually related to requirements. By using these manually defined links together with the dynamic program behavior logs from the execution of the test-cases, their tool is able to recover trace links between requirements and code. These trace links are derived using transitive reasoning and shared use of common ground. An example of transitive reasoning is when *A* depends on *B* and *B* depends on *C* then *A* depends on *C*. On the other hand, shared use of common ground is the use of the criterion that if *A* and *B* depend on subsets of a common ground (code in this case) and these subsets overlap, then *A* depends on *B*.

PRO-ART (Pohl, 1996a) is a prototypical requirements engineering environment, in which traceability relations are created as a result of creating, deleting or manipulating requirements. The focus of this work is on pre-requirements specification traceability. The authors propose a framework, which derives three dimensions of interest from the major sources of problems in requirements engineering, namely the *Representation Dimension*, the *Specification Dimension* and the *Agreement Dimension*. Based on the three-dimensional framework, a set of traceability models are developed to enable content oriented capture of the requirements engineering processes. The traceability models are formalized using the knowledge representation language Telos (Mylopoulos *et al.*, 1990). This work proposes the communication of actions (create, delete, modify) between tools. The communication between the actions is controlled by a communication manager, thus enabling process guidance and control. This approach assumes that interoperable tools are available for relating the inputs/outputs of the actions provided by the different tools as well as the requirements engineering environment is process centered

since the dependencies which are created between the inputs/outputs of different actions depend on the way the process was performed, i.e. on the sequence of actions.

Analysing existing trace relationships to obtain implied relationships may also provide a source of automatically recovered trace links. *TraceM* is a framework for automating the management of traceability relationships and it uses this technique. A key contribution of *TraceM* is its ability to transform implicit relationships into explicit relationships by processing chains of traceability relationships (Sherba *et al.*, 2003). This work builds on techniques from open hypermedia (Østerbye & Wiil, 1996) and information integration (Anderson *et al.*, 2002). *TraceM* allows stakeholders to view the explicit relationships in a system and to chain these relationships together to make previously implicit relationships explicit. Moreover, since these implicit relationships are being made explicit within the context of a comprehensive requirements traceability framework, they can be explicitly managed, tracked, and analyzed as the software project evolves.

Finally, trace information can be automatically recovered as by-products of transformation activities, that is, activities which automatically transform one artefact to another. This can be done either implicitly or explicitly (Olsen & Oldevik, 2007). By implicit, it is meant that a transformation engine generates the trace links automatically when a transformation is executed. By explicit, it is meant that additional code must be inserted into the transformation code in order to generate the trace information. A number of model transformation tools support the automated generation of trace-links. The ATLAS Transformation Language (ATL) (Jouault *et al.*, 2006) uses higher-order transformations (HOT) to add trace information in the transformation model. The Epsilon framework (Kolovos & Paige, 2010) also provides implicit traceability support through an external trace model that can be accessed in Epsilon's workflow mechanism (which is based on ANTLR). An Epsilon program, such as a transformation or a model merging operation, can expose trace information (in the form of a container of trace-links) and this information can be accessed by other model management tasks (such as validations) or even non-MDE tasks, such as visualisations generated with GraphViz.

In the MOF QVT Request for Proposals (RFP) issued by OMG in 2002, traceability is defined as an optional requirement (Object Management Group, 2011c). The specification describes three model transformation languages that can be used: Relations, Core, and Operational Mappings. In the Relations and Operational Mappings

languages, trace-links are created automatically without user intervention. In the Core language, a trace class must be specified explicitly for each transformation mapping. The QVT Operational implementation provided by Borland is one implementation that has this support. The implementation does not store trace models as external files, that would be inter-exchangeable between tools. This can be seen as a disadvantage of the tool (Kurtev *et al.*, 2007).

Similar solutions are provided by other model-to-model transformation languages and MDE frameworks such as Kermeta (Falleri *et al.*, 2006), FUJABA (Gorp & Janssens, 2005) and UniTI (Vanhooff *et al.*, 2007a)

Some existing model-to-text transformation languages have support for traceability as well. In the MOF Models-to-Text Standard (Object Management Group, 2006b), traceability is defined to be explicitly created by the use of a trace block inserted into the code. This approach provides user-defined blocks that represent a trace to the code generated by the block; this is specifically useful for adding traces to parts of the code that are not easily automated. A drawback of the approach is a cluttering of the transformation code. A complementary approach, as taken in MOFScript (Oldevik *et al.*, 2005), is to automate the generation of traces based on model element references. This approach is also taken in the Epsilon Generation Language (Rose *et al.*, 2008).

Acceleo Pro Traceability (OBEO07) is a traceability tool developed by Obeo that handles traceability links between model elements and code and vice versa. This tool enables round trip support; updates in the model or the code are reflected in the connected artefacts. Analyses are also available using the traces as input, but since this is a commercial tool, restricted information describing the solution is available. It seems to be based on similar ideas that are used in MOFScript where model elements are traced to exact positions in files.

Nevertheless, there are still open issues in the area of trace recovery. In the case of implicit trace recovery, the produced trace links do not have any case-specific semantics. They are generic and usually of the form $\langle source\ element, transformation\ rule, target\ element \rangle$. As will be discussed in more detail in the next section, such links do not facilitate any further analysis of the trace information. On the other hand, in the case of explicit trace recovery, trace links with rich-semantics can be generated. To achieve this, the traceability engineer has to insert traceability code into the transfor-

mation specification. Consequently, the transformation code becomes “*polluted*”, i.e. code of secondary importance is included, hence it becomes less readable.

2.4.2 Traceability Representation

The previous section presented the literature on the identification and recovery of trace links. In this section we focus on traceability representation. The traceability representation activity deals with the definition of the entity types to be traced and the relationship types between them. Moreover, it deals with the representation of the captured traceability information in the form of data structures as well as their visualisation and storage.

2.4.2.1 Specification of Traceability Information

Many different stakeholders such as project managers, analysts, designers, testers, and end users are involved in the system development life cycle. The traceability needs of these stakeholders differ due to differences in their goals and perspectives. Consequently, there are wide variations in the format and content of traceability information across different system development efforts. Providing traceability support during system development can be along several dimensions. For example, the types of links of interest to a system tester may be different from those of interest to a system designer. A system tester may be interested in finding out what are the system components that are affected by a requirement while a system designer may be interested in finding out how the components of the system are affected by requirements. Moreover, the semantics of a trace link as viewed by different stakeholders may differ. In the case of a *requirement-to-component* trace link for example, the system designer might consider the requirement as a constraint on the system design, while a system tester might be interested in using the traces to perform coverage analysis in order to find out if every requirement is implemented by at least one component. Therefore, the meaning (semantics) of trace links is guided by the reasoning that the trace user will be performing with the relationship (Ramesh & Jarke, 2001).

Many researchers have acknowledged the need to distinguish between different trace link types with specific semantics in order to facilitate richer traceability analysis. (Bayerand & Widen, 2001) propose that trace links should have rich semantics in

order to compensate for the high cost of establishing and maintaining traceability, while (Pineiro & Goguen, 1996) suggest that trace links should be precisely defined in order to avoid the problem of culture-based representations. Similarly, (Dick, 2005) advocates the use of rich traceability semantics in industrial settings.

To address the aforesaid issues associated with the representation of trace links, researchers have attempted to define the syntax and semantics of traceability information. To do so, they have constructed specifications of what constitutes a valid traceability model. These specifications are called *metamodels*, *schemes*, *traceability classifications* or *reference models* depending on the field in which they are used. For the purposes of this work, we will refer to this kind of specifications as classifications. Traceability classifications have been developed that emphasise different attributes, characteristics, and viewpoints (ranging from conceptual models to concrete designs that can form the basis of an implementation) on traceability. For example, some classifications are based on the types of the related artefacts, while others are based on the use of traceability in supporting different development activities (Spanoudakis & Zisman, 2005).

There are two types of traceability classifications - generic or case-specific. In the case of generic classifications, the main assumption is that a generic and complete set of traceability standards can be found and described. This set will determine the types of trace links, semantic expressiveness and other qualities of traces that should be recorded in every case. On the other hand, the advocates of the case-specific classifications argue that such a generic set of standards can not be defined, since the space of trace links and of their constraints is vast and most of the time the decision of which link types to use depends on the particular usage scenario as well as on the domain. The contributions in this area have to do mainly with mechanisms of how to ease the definition of the required traceability information for every case-specific scenario.

Generic Traceability Classifications One of the main classifications in this category is the one proposed by (Ramesh & Jarke, 2001), who investigated quite exhaustively the completeness of a traceability classification in the domain of requirements engineering. To do so, they made extensive observations in several industrial projects over a number of years. The results of their observations are captured in a set of reference models which consist of the most important kinds of traceability links for the various development tasks. A basic traceability metamodel is derived and it is illustrated in Figure

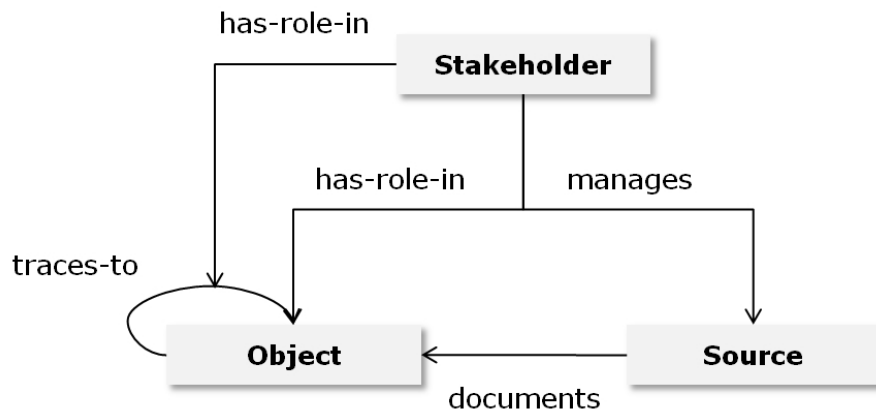


Figure 2.5: Traceability Metamodel according to (Ramesh & Jarke, 2001)

2.5. *Sources* represent the people who are involved in the development process. *Objects* stand for the traceable entities, while *Sources* correspond to sources of information or knowledge such as documents, standards, meeting minutes, policies, etc. The three entities of the meta model correspond roughly to the three dimensions of requirements engineering proposed by (Pohl, 1996a) in that they cover the aspects of *understanding* (objects), *agreement* (stakeholders), and *physical representation* (sources).

The proposed metamodel can be used to represent six dimensions of traceability information.

- **What** information should be recorded as traceability information?
- **Who** are the Stakeholders who create, update and use the traceability information as well as the traceable entities?
- **Where** does the traceability information come from?
- **How** is traceability information represented? **How** does this information relate to other traceability components?
- **Why** has a traceable entity been created, modified or evolved?
- **When** has the traceability been captured, modified or evolved?

Following (Ramesh & Jarke, 2001), there are two types of traceability users - *low-end* and *high-end* users. Low-end users consider traceability simply as a mandate from the project sponsors or for compliance with standards. These users describe the various project interdependencies using simple traceability classifications. Conversely, high-end users consider traceability as a major opportunity for customer satisfaction and knowledge creation throughout the systems development process. Therefore, they use much richer traceability classifications, which enable them to perform richer analysis and reasoning on traces. (Ramesh & Jarke, 2001) propose two levels of reference models depending on the type of the traceability users. The first level is aimed at the low-end users and it provides just a handful of trace link types. The second level is aimed at the high-end users and it consists of approximately 50 different types of trace links. These trace links include relationships such as *satisfies*, *describes* and *depends on*.

Another classification focusing on requirements traceability is proposed by (Pohl, 1996b). In this work, 18 different trace link types are identified and organised into five different categories. These categories are the following:

- *Condition Link Group*: this group consists of the relationships between requirements and the various constraints associated with them.
- *Documentation Link Group*: this group includes relationships between different types of software documentation and requirements.
- *Abstraction Link Group*: this category includes relationships representing abstraction between requirements, such as generalisation or refinement.
- *Evolutionary Link Group*: this group consists of replacement relations between requirements.
- *Content Link Group*: this category includes trace links which denote comparison, conflict and contradiction between requirements.

A similar approach is proposed by (Spanoudakis & Zisman, 2005), who has conducted a thorough literature survey in order to define 8 categories of trace link types. These categories are the following:

2.4 Traceability Related Activities

- *Dependency* links describe a relationship in which an element e_1 *depends* on an element e_2 , if its existence relies on the existence of e_2 or if changes in e_2 have to be reflected in e_1 .
- *Generalisation/Refinement* links are used to identify decomposition of entities, composition of entities or refinements.
- *Evolution* links are used to describe the *evolution* relation between two artefacts. When artefact e_1 evolves to artefact e_2 , then artefact e_1 is replaced by e_2 .
- *Satisfiability* links signifies the compliance of an artefact to another one.
- *Overlap* links describe a relationship in which artefacts e_1 and e_2 refer to common features of a system or of its domain.
- *Conflict* links describe an incompatibility or an inconsistency relation between two artefacts.
- *Rationalisation* links are used to represent the rationale behind the creation and evolution of artefacts.
- *Contribution* links represent the association between artefacts and stakeholders, who have contributed to their creation.

In addition to the aforementioned trace link categories, (Spanoudakis & Zisman, 2005) present a very detailed description (in the form of a matrix) of the various trace link types that have been proposed in the literature and the types of software artefacts that these links interrelate. Moreover, associations between stakeholders and artefacts are presented. Based on their matrix description, they observe that most approaches focus mainly on types of traceability relations that relate requirements specifications, as well as requirements with design. The number of approaches, which focus on the links between code and requirements or between code and design artefacts are far less. This is attributed to the fact that initially, traceability was a concept very closely related to requirements, as well as to the fact that establishing trace links between code and other development artefacts is a hard task. Finally, (Spanoudakis & Zisman, 2005) identify the lack of standard semantics for the various trace link types found in the literature and

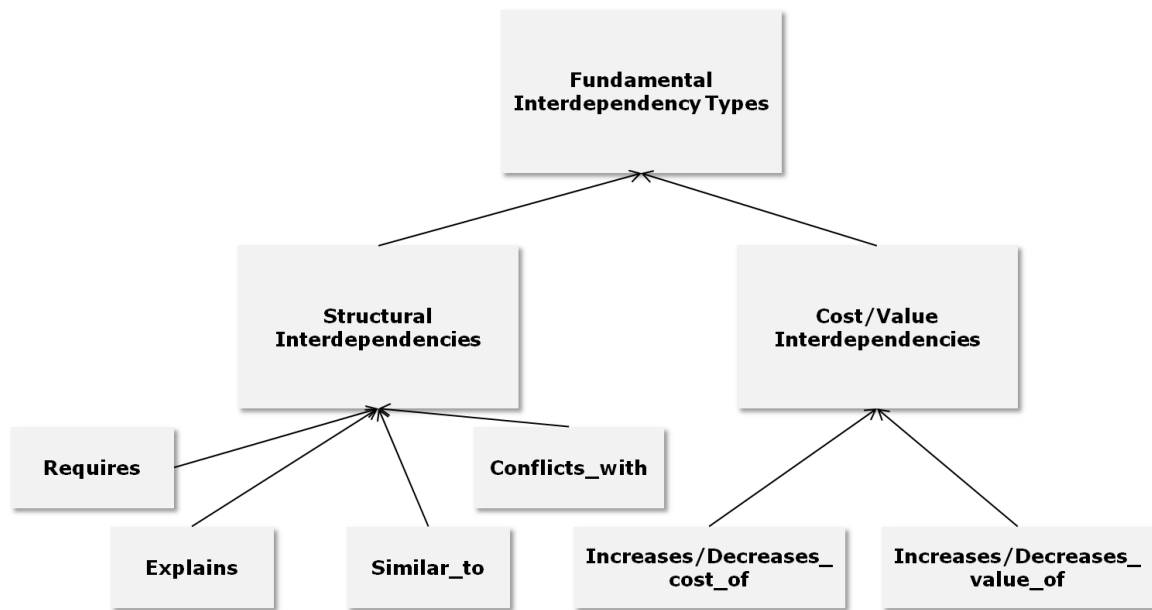


Figure 2.6: Requirements Interdependencies Classification (Dahlstedt & Persson, 2003)

they stress the need for richer trace link semantics, since it is the precondition for the development of tools for the generation, maintenance and deployment of meaningful trace links.

(Dahlstedt & Persson, 2003) propose a classification of interdependency trace links between requirements. This classification is illustrated in Figure 2.6. In the proposed classification, requirements interdependencies are grouped into two main categories - structural and cost/value interdependencies. Structural interdependencies are concerned with the fact that given a specific set of requirements, they can be organised in a structure where relationships are of a hierarchical nature as well as of a cross-structure nature. On the other hand, cost/value interdependencies are concerned with the costs involved in implementing a requirement relative to the value that the fulfillment of that requirement will provide to the perceived customer/user.

Another classification of trace link types is proposed by (von Knethen & Paech, 2002). This classification investigates the technological aspects of traceability in five different dimensions. One of the proposed dimensions for classifying traceability approaches is the types of relationships they support. That is, what kinds of relationships

are described by trace links, what is their direction, what are their attributes, what is the setting of those relationships and finally how these relationships are represented. From the literature the authors identify three general kinds of relationships: (1) relationships between documentation entities at different levels of abstraction, (2) relationships between documentation entities on the same abstraction level, and (3) relationships between documentation entities of different versions of the same software product.

There are a few open issues with the traceability classifications mentioned above. The semantics of the link types they describe is not formally defined and it is subject to the view and interpretation of the reader. Furthermore, most of the link types described in those classification are binary, something which comes to contrast with the fact that trace links can very often be n-ary (Munson & Nguyen, 2005). (Dick, 2005) advocates the need for even more complex trace link structures such as alternatives and conjunctions.

In an attempt to specify the semantics and the composition of traceability information in a well-structured manner, the MDE community encodes such information in well-defined metamodels.

(Walderhaug *et al.*, 2006) propose a traceability metamodel, which sets a basis for defining trace models. This metamodel consists of four entities. The *TraceModel* entity is used to represent the container for the various trace links as well as for the various traceable artefacts. The *TraceableArtefactType* entity defines the mapping of a specific model artefact type to a corresponding traceable artefact. The *ArtefactTraceType* defines a specific trace type for a *TraceableArtefactType*. Finally, the *RelationTraceType* defines a specific trace type for a certain relation between a source and a target artefact type. Instantiations of this metamodel can be used to model different traceability scenarios.

(Amar *et al.*, 2008) propose a traceability metamodel which is suited to model transformations. Their metamodel allows to structure the traces, which are generated by the transformation engine. The proposed metamodel is an extension of the one of (Falleri *et al.*, 2006). The authors argue that it is useful, for imperative as well as declarative transformations, to have a multiscaled trace. The fact that an operation transformation can call another one (or that the rules can trigger other rules) creates levels of nesting that it's useful to be able to represent. That's why the composite pattern is applied on the trace links. This metamodel is illustrated in Figure 2.7.

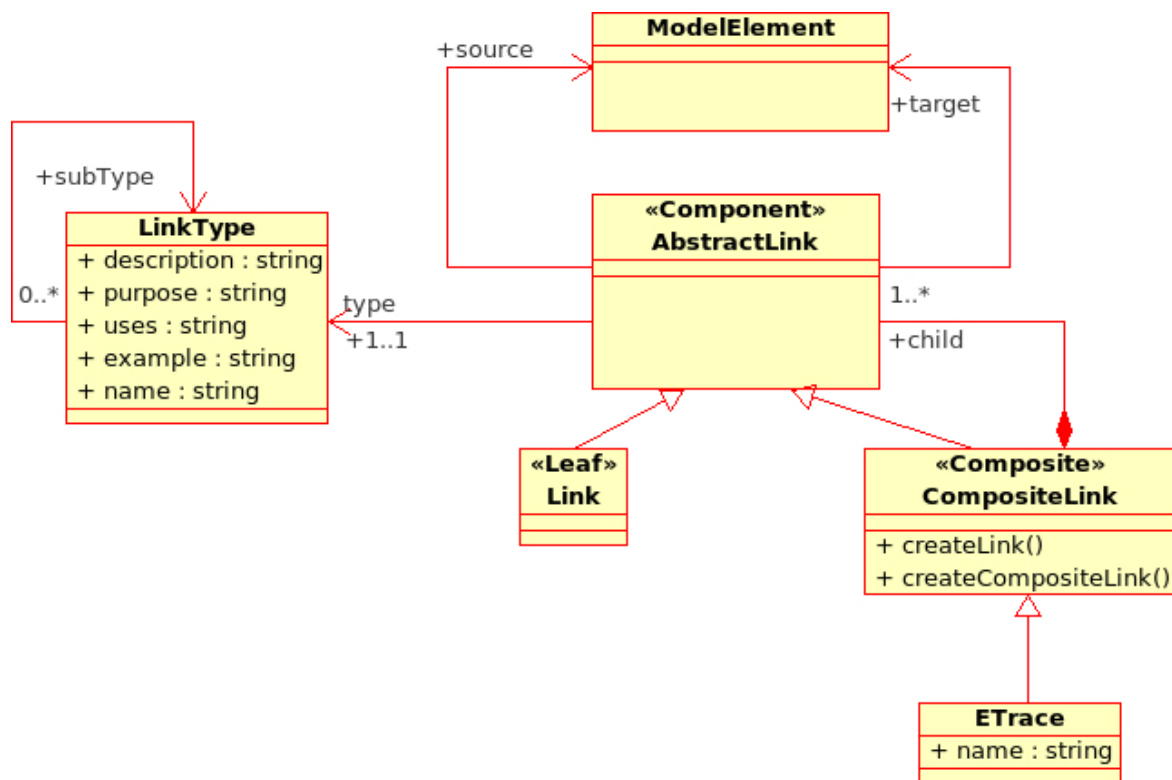


Figure 2.7: Traceability Metamodel proposed by (Amar *et al.*, 2008)

Similarly to (Amar *et al.*, 2008), (Jouault, 2005) aims to provide traceability support for model transformation scenarios. Towards this goal, he proposes a concise traceability metamodel (Figure 2.8). It consists of two entities, the *TraceLink* entity and the *AnyModelElement* entity. The *TraceLink* entity has a *ruleName* attribute, which stores the transformation rule name and pointing to *AnyModelElement* via two multivalued references: *sourceElements* and *targetElements*.

Another traceability metamodel can be found in the work done by (Sousa *et al.*, 2008). Apart from the metamodel entities which are encountered in the metamodels presented above such as trace link, traceable artefact and traceable artefact type, additional metamodel entities are used. Each link has a link type as well as a scope which describes for which entities a link is valid.

Several other researchers from the MDE community (e.g. (Barbero *et al.*, 2007; Grammel & Voigt, 2009; Olsen & Oldevik, 2007; Vanhooff *et al.*, 2007b)) have come

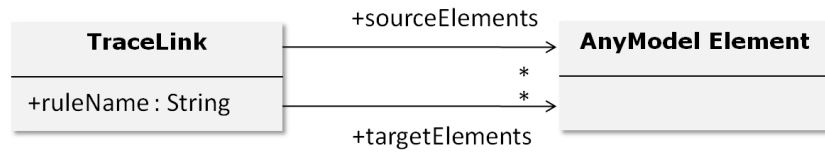


Figure 2.8: Traceability Metamodel proposed by (Jouault, 2005)

up with their own traceability metamodels. These metamodels usually extend a basic traceability metamodel with case-specific enhancements. A traceability metamodel which is generic and adequate for every traceability scenario is yet to be found (Limon & Garbajosa, 2005). Hence, another research direction on traceability representation focuses on ways to facilitate the definition of case-specific traceability information. This strand of research is presented in the next section.

Case-specific Traceability Classifications The space of trace-links is vast; many different kinds of case- and domain-specific trace-links have been identified. These are often presented in the form of *traceability classifications*; a number of these have appeared in the literature. For example, classifications given in terms of scenarios of use of traceability are postulated by (Olsen & Oldevik, 2007; Walderhaug *et al.*, 2006). Classifications in terms of specific domains have been produced by (Ramesh & Jarke, 2001) for requirements engineering, for business applications (Rummler *et al.*, 2007), and for usability and accessibility (Power *et al.*, 2009). Moreover, traceability classifications in MDE have been developed that emphasise different attributes, characteristics, and viewpoints (ranging from conceptual models to concrete designs that can form the basis of an implementation) on traceability. It is apparent from the above, that different stakeholders in different domains have different needs in regards to traceability. Hence, different traceability classifications are needed.

(Paige *et al.*, 2008) propose a lightweight process for engineering traceability classifications from different viewpoints. The lightweight process is called the Traceability Elicitation and Analysis Process (TEAP). It is derived from a process developed in (Chan & Paige, 2005) for eliciting and understanding different forms of model-based contracts. The aim of TEAP is to elicit and analyse traceability relationships in order to determine how they fit into a traceability classification. While eliciting new traceability relationships, we improve our understanding of the key attributes of these traceability

2.4 Traceability Related Activities

relationships: the artefacts they involve, their semantics, and their domain of applicability. TEAP is an iterative process. There are three main activities: Elicitation, Analysis, and Classification, outlined in Table 2.1. Elicitation involves studying the domain and available scenarios to help identify new trace-links; Analysis involves developing our understanding of the trace-links' semantics and their relationship to other trace-links; and Classification involves structuring the trace-links.

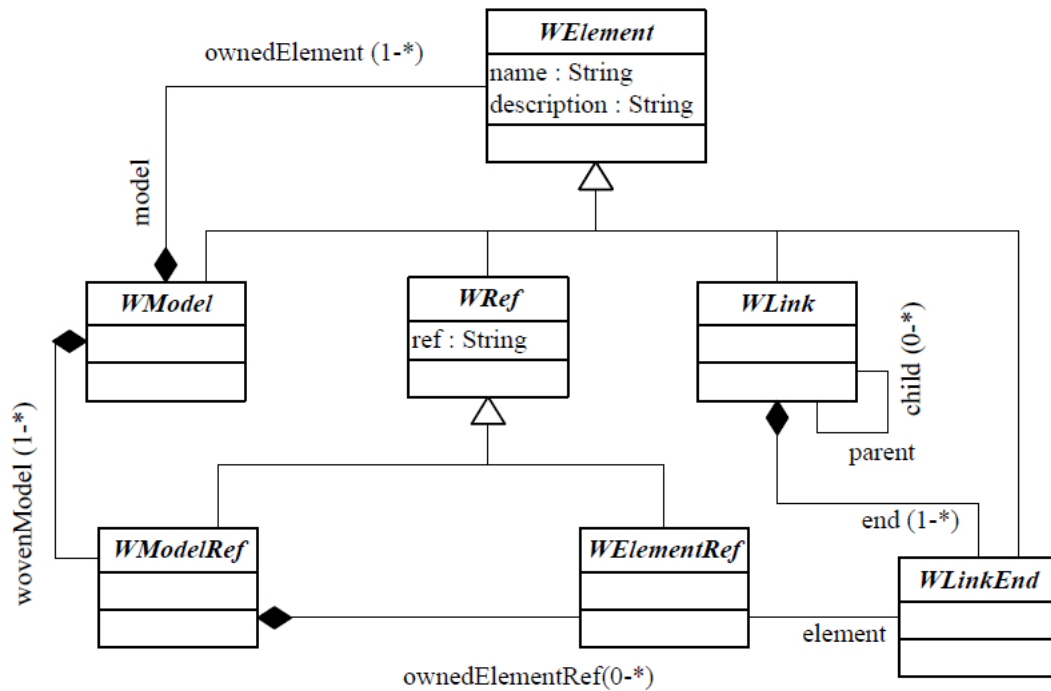
Activity	Description
Elicitation	Identify new types of trace-links and relationships.
Analysis	Understand relationships between new trace-links and existing trace-links; identify constraints.
Classification	Build and/or refine a classification.

Table 2.1: TEAP activities

TEAP deals mainly with understanding a domain's traceability needs. The traceability engineers are responsible then to encode their understanding, usually in the form of a metamodel. This is a technical task and it is not addressed by TEAP. (Fabro *et al.*, 2005) propose a tool which supports this activity. The Atlas Model Weaver (AMW) can capture and store links between models of diverse metamodels. The various links are stored in models, which are called *weaving* models. Weaving models have special characteristics. They are not self contained, i.e., a weaving model is useful only if the related models exist as well. To support link semantics, the user of the tool can extend the core weaving metamodel (Figure 2.9) with case-specific semantics. To extend the core metamodel, the user writes a Kernel Meta Meta Model (KM3) (Jouault & Bézivin, 2006) script in order to specify the case-specific link entities and then the tool merges automatically these link entities with the core weaving metamodel.

(Guerra *et al.*, 2010) propose an *inter-modelling* language, which allows expressing relations between models in a declarative way, using both structural object patterns and declarative attribute conditions. This language can express relationships between models, which can be then used in different scenarios. Model-to-model traceability is one of those scenarios.

While there is a significant amount of work in the area of generic classifications, during the conducted literature review we have not encountered any other approaches on

Figure 2.9: AMW core metamodel by (Fabro *et al.*, 2005)

case-specific traceability classifications. One possible reason might be the fact that only recently have researchers started investigating ways to support the creation of case- or project-specific traceability classifications instead of trying to find a generic traceability metamodel which applies in any traceability scenario.

2.4.2.2 Actualisation and Visualisation of Traceability Information

Deciding on which traceability information is relevant and meaningful is only one activity associated with traceability representation. Once such decision is made, traceability information have to be realised in an appropriate form in order to be used in software projects. We call this activity *traceability actualisation*. Moreover, since the traceability information captured for a particular project can be very complex, appropriate visualisation techniques have to be used in order for this information to be presented to their intended users.

2.4 Traceability Related Activities

(Marcus *et al.*, 2005) argue that visualising traceability links is important, non trivial, and considerable support is needed. Following (Wieringa, 1995), there are three different ways to realise and visualise traceability information. This can be done by using traceability matrices, cross-references or graphs (models or diagrams).

A *traceability matrix* is a simple two-dimensional grid in which the horizontal and vertical dimension list the items that can be linked, and the entries in the matrix represent links between these items. Table 2.2 illustrates a sample traceability matrix, whose purpose is to store traceability relations between requirements and test cases. When a test case is associated with a particular requirement then an x at the intersection of the corresponding row and column indicates this relationship.

Req. Identifiers	Req. UC 1.2	Req. UC 1.3	Req. UC 1.4	Req. UC 1.4	Req. UC 1.5
Test Cases	2	2	2	1	0
1.1.1	x				
1.1.2	x	x			
1.1.3		x	x		
1.1.4			x		
1.1.5				x	

Table 2.2: Sample Traceability Matrix

According to (Wieringa, 1995), the matrix representation can be considered equivalent to a graph representation, but it is less orderly as a visual representation technique. Traceability matrices can be easily understood and used by non-technical users. Moreover, they can very efficiently capture simple traceability relations.

However, there are open issues with traceability matrices that have to do with the complexity and scale of real-life projects. First, only binary relations between entities can be represented. However, very often n-ary relationships are required (Sherba *et al.*, 2003). Another important limitation of traceability matrices is the fact that rich link semantics can not be represented easily. That is, different link types and project specific constraints can not be captured by this simple form of traceability information representation. Finally, since traceability matrices are in tabular form, they are typically created using a spreadsheet application and are independent of the artefacts they refer to. Hence, the process of capturing and maintaining traceability is mainly done manually.

Nonetheless, in (Cleland-Huang *et al.*, 2003), the authors argue that the number of traceability links that need to be captured grows exponentially with the size and complexity of the software system. This means that manually capturing traceability data for large software projects requires an extreme amount of time and effort. In an attempt to address the aforesaid limitations, researchers have come up with more sophisticated traceability matrices. For example, for different types of links, different symbols or colours can be used (Pinheiro, 2004).

A different way to realise and visualise traceability information is by treating artefacts as nodes and the trace links between them as edges of a *traceability graph*. Such graphs or models are used in several approaches. In (Fabro *et al.*, 2005), traceability models, which are considered as a particular case of weaving models, are used by the AMW tool. Several other approaches use diagrams to represent traceability. These approaches include the work done by (Göknil *et al.*, 2010; Pinheiro & Goguen, 1996; Pohl, 1996a; Ramesh, 2007).

Following (Kolovos *et al.*, 2006b), there are two strategies to storing and managing traceability information in the case of model-to-model traceability. In the first one, the traceability information is embedded in the models they refer to, while in the second strategy the traceability information is stored separately from the models.

In the intra-model storage of traceability information case, the traceability information is stored within the artefact they refer to in the form of model elements or model element attributes, such as tags and properties. Despite its simplicity and human friendliness, keeping such information with the artefacts can be problematic for several reasons. If the link is directed and stored in the source model only, it is not visible from the target model. On the other hand, if the traceability information is stored in both models, then this information must be maintained consistent, thus the burden of maintaining consistency is introduced every time a change occurs (Aizenbud-Reshef *et al.*, 2006b). In addition, embedding traceability information inside a model causes “*pollution*” (Kolovos *et al.*, 2006b), i.e. the inclusion of elements in the model of secondary importance. Such an inclusion can render a model overcrowded and can make it difficult to understand and maintain. Finally, the issue of *uniformity* arises in this approach (Oldvik & Aagedal, 2005). In an MDE environment, it is common that models have their own representations and semantics. Hence, it is very difficult to distinguish the traceability information from the other model artefacts. As a result, automated analysis of

traceability information becomes very challenging. The main approaches falling under this strategy utilise mainly language specific constructs. For example, specific types of traceability links are represented in UML diagrams by using stereotyped dependencies, such as $\ll \textit{refines} \gg$ (Heaven & Finkelstein, 2004).

In the second strategy for representing traceability, traceability information is stored separately from the artefacts they refer to in a separate model. Constructing such models has two clear advantages. First, the source and target models remain “clean”, since the traceability links are stored in a separate model, whose concern is to capture this kind of information. In this way, the aforementioned “pollution” is avoided. In addition, storing traceability links in a model who conforms to a metamodel with clearly defined semantics makes automatic analysis by tools much easier. A prerequisite for storing traceability links externally from the models they refer to, is that the various model elements have unique identifiers, so that the related traceability links can be resolved unambiguously (Kolovos *et al.*, 2006b). For example, such a mechanism is provided by MetaObject Facility (MOF) (Object Management Group, 2006a) and Eclipse Modelling Framework (EMF) (Eclipse Foundation, 2010a) in the form of a *xmi.id* identifier.

The main drawback of using diagrams as a method of traceability representation and visualisation has to do with human’s ability to represent and read complex structures on two or even three dimensions. Depending on the complexity and size of the traceability scenario, diagrams can become cluttered and difficult to read very easily (Herman *et al.*, 2000). Thus, using abstraction and separation of concerns to represent complex traceability scenarios is a common practice.

The third strategy for realising and visualising traceability information is by using cross-references. Cross-references can be considered as pointers which are embedded in an artefacts and point to another artefact with which there is a relation. These pointers can be written in natural language or in more sophisticated ways such as hyperlinks.

Using cross-references to represent trace links can be intuitive and easy. Moreover, there is software which can generated reports out of a document’s references (Wieringa, 1995). However, cross-referencing does not provide a concise representation of trace links to and from a document. Moreover, cross-references are always binary links. As mentioned above, in realistic traceability scenarios the need for n-ary relations arises very often. Finally, a cross reference can indicate only the existence of a relation be-

tween two artefacts, but it provides no information for the type of the relation or the rationale of its existence.

2.4.3 Traceability Maintenance

The third main activity associated with software traceability is *traceability maintenance*. Traceability maintenance is very closely related to software entropy (Bianchi *et al.*, 2001). Software entropy has to do with the increase of disorder of software as it is evolved. According to the second law of thermodynamics, a closed system's disorder cannot be reduced, it can only remain unchanged or increase. A measure of this disorder is entropy. This law seems plausible for software and its associated artefacts, such as documentation or design. As software is modified, its disorder always increases.

Since traceability information is tightly coupled with the other artefacts of the development process, it is subject to software entropy as well. That is, traceability information is subject to gradual degradation as the related artefacts are modified. As a result, the recorded relationships may end up being incorrect or inaccurate and as a result cannot support change propagation or impact analysis activities. One of the most challenging aspects of traceability is how to maintain the integrity of the relationships, i.e. trace links, while the referenced entities continue to change and evolve. To achieve this, referential integrity and link integrity must not be violated. The difference between referential and link integrity is very subtle. In the context of relationship management, referential integrity is a measure of the reliability of a reference to an end-point, whether a source or a destination of the relationship. On the other hand, link integrity measures the reliability of the whole link (i.e. all endpoints) (Davis, 1998). When the entity at the end of a link is not present or is not the entity that was intended by the link author, then the referential integrity is violated and the link is said to be *dangling*. Both manual effort and computation time need to be invested to ensure referential integrity and the goal is to minimize manual effort at the expense of computational time. Ensuring referential and link integrity requires the examination of both the links and the changes of the models under consideration.

There are two main approaches for maintaining trace link integrity: event-driven and state-based approaches. In the former approach, the elementary changes of the various model elements are constantly monitored, change events are generated based

upon these elementary changes and finally according to the change events corrective actions are taken. In the latter approach, the detection of model changes takes place by comparing different versions of the models and *suspect* links are found based on the identified changes. In the following, the main contributions in those two research areas are presented.

2.4.3.1 Event-Driven Traceability Maintenance

Event-driven traceability maintenance approaches resolve around the monitoring of elementary artefact change events and the generation of compound changes. This is achieved by utilizing a set of rules for recognizing the events as constituent parts of intentional development activities. Once these activities have been identified, traceability links related to the changing model elements can be updated automatically.

The atomic model changes associated with the first phase of this approach are usually addition, deletion and modification of artefacts. In the second phase, the development activity that is realized by a chain of the aforesaid changes must be recognized. Such activities include replacement, merging and splitting of traceable artefacts. Atomic changes are associated to development activities using a predefined set of rules.

There are three main challenges associated with recognizing the various compound development activities (Mäder *et al.*, 2008). First, the same development activity can be achieved by different elementary changes. For example there are two different ways to replace a model element. The developer can either delete the original element and then add the new one or she can just modify the existing one. An additional challenge is the fact that the same development activity can be achieved by the same elementary changes in different sequences. If for example a developer wants to replace a model element, she can either add the new one first and then delete the old one or vice-versa. Finally, the type of change and the impacted model element do not offer enough information for relating changes to each other. The final step of the event-driven approaches to traceability maintenance is the reconciliation of the dangling links. This can be performed manually by the user or automatically if a predefined rule exists for every change event.

To overcome the identified challenges, the event-driven approaches restrict their scope in two main ways. First, the user is restricted to using a particular tool for evolving the artefacts. For example, in the event-driven approach proposed by (Cleland-Huang

et al., 2002a) for maintaining trace links between requirements and other development artefacts, requirements have to evolve using a specific editor and a particular notation. In this tool, to merge two requirements r_1 and r_2 the user has to write the following expression:

$$r_3 \rightarrow r_1 + r_2 \quad (2.1)$$

Although, this tool is effective for tracking requirement changes, it can not be used in the general case since different artefacts will evolve differently and they will need different editors.

The second restriction which is imposed by many even-driven approaches has to do with the notations or artefacts they support. Such approaches usually focus to only one type of artefact or one notation and usually they use specific characteristics of the notation in order to identify the compound changes. For example, according to the UML specification a bidirectional association between two entities can be considered equivalent to two unidirectional associations between the same artefacts. This is a characteristic of the UML notation, which can be used by an event-driven tool to identify this refinement and adjust traceability information accordingly. However, this rule can not be generalised since it is UML specific.

Since event-driven approaches focus on a particular domain, artefact or notation, they can preserve link integrity to a satisfactory degree. However, due to the multitude of different traceability scenarios, a more generic solution to the problem of traceability maintenance is required. Such a solution is yet to be found.

One of the most influential and well-known event-driven approaches is the one proposed by (Cleland-Huang *et al.*, 2002a). In this approach, a publish-subscribe mechanism is used. This mechanism follows the *Observer* design pattern. An interrelationship between a requirement and another artefact is register to a central server. The evolution of requirements is then expressed as a series of change events. When a requirement is changed, all *observers*, i.e. artefacts which are related to this requirement, are notified about this change. The stakeholders who are responsible for the maintenance of the aforementioned observers could then check for potential changes in the traceability links. The change in requirements is performed using a dedicated to this purpose notation, hence the identification of complex events such as decomposition or replacement of requirements is possible.

Another event-based approach to traceability maintenance is proposed by (Mäder *et al.*, 2009). TtraceMaintainer is an extension to UML tools, which records operations performed by a developer. If a chain of operations is identified as a single change event, then the tool is able to maintain trace links which are affected based on predefined policies. If no policy is defined then the user is notified about the change. TraceMaintainer works only with UML structural models, though the authors investigate the applicability of their approach in other types of models.

A similar approach to traceMaintainer is the one proposed by (Murta *et al.*, 2006). ArchTrace is a tool that addresses the consistency and evolution of trace links between software architecture models and their associated code. ArchTrace relies on an infrastructure for identifying change events in the architecture models and continuously updating the trace links based on the change events and a set of policies. These policies are atomic elements which can be disabled and enabled individually so that they fit to the user's situational needs. The ArchTrace tool assumes the use of xADL 2.0 (Dashofy *et al.*, 2001) to describe software architectures and Subversion (Collins-Sussman *et al.*, 2004) to store source code.

Finally, (Aizenbud-Reshef *et al.*, 2005) define the operational semantics of traceability in UML. One of the questions they try to answer is what actions should be triggered when an event occurs which impacts on the traceability relationship in order to ensure that the relationship is still valid according to its semantics. To this end, they define two types of semantics for traceability, namely *preventative* and *reactive* semantics. The former type describes actions which should not happen, such as deleting an entity which is dependent on another entity before the dependency is deleted. For example, if a class A imports a class B, class B can not be deleted before the import dependency is removed. The latter type of traceability semantics describes what actions should be triggered when an event occurs which affects the referenced entities or the traceability relationship itself. To support operational semantics for traceability c argue that a set of semantic properties should be defined. Each semantic property is a triplet of the form:

$$\{event, condition, actions\} \quad (2.2)$$

An *event* involves an element of the relationship, a *condition* is a logical constraint of the relationship and *actions* can be either preventative or reactive. One of the main

shortcomings of this solution is the fact that they only specify three atomic events - create, modify and delete. However, they do not propose what should happen with more complex events such as merge or split actions. Additionally, they do not specify how the various events should be recognized in order for the actions to take place.

2.4.3.2 State-based Traceability Maintenance

In this method, the detection of model changes takes place by comparing an instance of the model under consideration in time t_1 with an instance of the model in time t_0 , where t_0 is the time when the model was checked for the last time. This comparison can take many different forms. Usually, all the artefacts under consideration are expressed in a common representation, such as XMI, and then their *diff* is calculated. Change detection can take place either in predefined times (for example when a model is saved) or when the user requires to ensure that the trace link integrity is not violated. If model changes are detected, the trace links, which refer to those models should be updated. If there are predefined policies associated with the detected changes, then the link maintenance can be done automatically. State-based approaches are often limited in detecting only syntactic model changes, while semantic model changes can jeopardise the link integrity as well. To our knowledge, an approach which addresses this issue does not exist up till now.

An example of a state-based approach is the one proposed by (Sharif & Maletic, 2007). In this approach a difference tool such as EMFCompare is used to identify syntactic differences between different versions of a model. Based on these differences and user input the links are evolved. (Maletic *et al.*, 2005) propose an analogous approach. Text differencing is used to identify syntactic changes in different versions of source code or of XML representations and according to the identified changes user input is required to reconcile the dangling trace links.

2.4.4 Traceability Usage

Recording and maintaining traceability is not an end in itself. On the contrary, traceability is deployed in the development life cycle of a software system to support different development and maintenance activities. Following (Aizenbud-Reshef *et al.*, 2005), the type of traceability information captured for the needs of a project depends mainly on

the project's traceability goals, i.e. the intended use of the traceability information. Reviews of different traceability usage scenarios are presented in (Gotel & Finkelstein, 1994; Ramesh & Edwards, 1993; von Knethen & Paech, 2002). The main goal of this section is to identify and briefly describe the various traceability usage scenarios reported in the traceability literature.

2.4.4.1 Traceability for change management and impact analysis

One of the primary motivations for capturing traceability information between various artefacts of the development lifecycle is the ability to use the established relationships in order to assess the potential impact of changes in one part of the system to other parts of it (*change impact analysis*). The ability to determine the impact of a change improves the maintainability of a system according to many researchers such as (Arkley & Riddle, 2005; Neal, 1994). Moreover, traceability information can help the engineers make decisions about whether such changes should be introduced and with which priority (*change management*) (Spanoudakis & Zisman, 2005).

Primary change impact analysis consists of identifying particular traceability relation types of interest as well as possible attributes associated with them. In this simple form of impact analysis, all the artefacts which are associated to the artefact which is changed are identified and the possibility of their being affected by this change is assessed based on the type of relationship they have with the changed artefact.

(Spanoudakis & Zisman, 2005) argue that more complex forms of impact analysis might be needed depending on the scenario. Examples of such scenarios include the classification of affected artefacts into different groups depending on the effect the change will have on them or the estimation of the cost of propagating the change. To support such complex scenarios, semantically-rich traceability must be established between the various artefacts. The necessity of having rich traceability semantics in order to use traceability for impact analysis in complex scenarios has been identified by empirical studies (Bianchi *et al.*, 2000; Sandahl, 1996). Moreover, the granularity of the relationships affects the results of impact analysis (Bianchi *et al.*, 2000).

Traceability for impact analysis purposes has been proposed by many authors. (Gotel & Finkelstein, 1994) and (Ramesh & Edwards, 1993) propose the use of trace links

to trace changes of requirements to downstream artefacts. In such cases, trace information could be used to assess the impact of the change on implementation time as well as on implementation costs. Furthermore, (Arkley & Riddle, 2005) propose the use of trace links to predict the impact of requirements changes on the quality of a system, such as the stability of it. (von Knethen & Grund, 2003) have developed a tool named *Quatrace*, which can be used for semi-automatic impact analysis based on traces. (O'Sullivan, 2003) came up with a UML model-based approach to impact analysis that can be applied before any implementation of artefact changes, thus allowing an early decision-making and change planning process.

Finally, traceability information can be used to propagate changes between related artefacts so that those artefacts remain synchronised and consistent. (Fritzsche *et al.*, 2008) discuss how trace links can be used to relate performance models with application models in order to provide to the developers performance prediction feedback mechanisms.

2.4.4.2 Traceability for V&V activities

Trace links can provide the basis for performing validation and verification analysis. By validation, it is meant the analysis undertaken to ensure that a system fulfills its intended purpose. On the other hand, by verification it is meant the analysis undertaken to ensure that a system is build according to its specification. Traceability is mandated in several standards like DO-178b (for Aeronautics , RTCA) fof V&V purposes.

In (von Knethen *et al.*, 2002), preliminary system verification takes place by *composing dependency* and *satisfiability relations* in order to establish whether all the requirements of a system have been allocated to specific design or source code components. Moreover, in (Gotel & Finkelstein, 1995) *contribution* trace links are used to identify stakeholders and involve them in requirements validation activities. Finally, (Ramesh & Edwards, 1993) uses traceability information to detect inconsistency, incompleteness and other defects of requirements specifications.

For the purpose of system verification, (Fiutem & Antoniol, 1998) present an approach to check the compliance of OO design with respect to source code using traceability information. Similarly, (Spanoudakis & Kim, 2004) propose techniques for detecting *overlap* relationships between structural and behavioral object-oriented models

of software system and then they use this information to check whether the overlapping elements satisfy specific consistency rules and. In cases where these rules are violated, the proposed approach guides software designers in handling the detected inconsistencies.

2.4.4.3 Traceability for Testing

In addition to the aforementioned usage scenarios, traceability relationships can be used to check the existence of appropriate test cases for testing different requirements. The results of such analysis usually provide input to software inspection and auditing activities (von Knethen *et al.*, 2002). Furthermore, traceability can be used to related possible solutions for failed tests to the actual problems (Figure 2.10) (Arkley & Riddle, 2005).

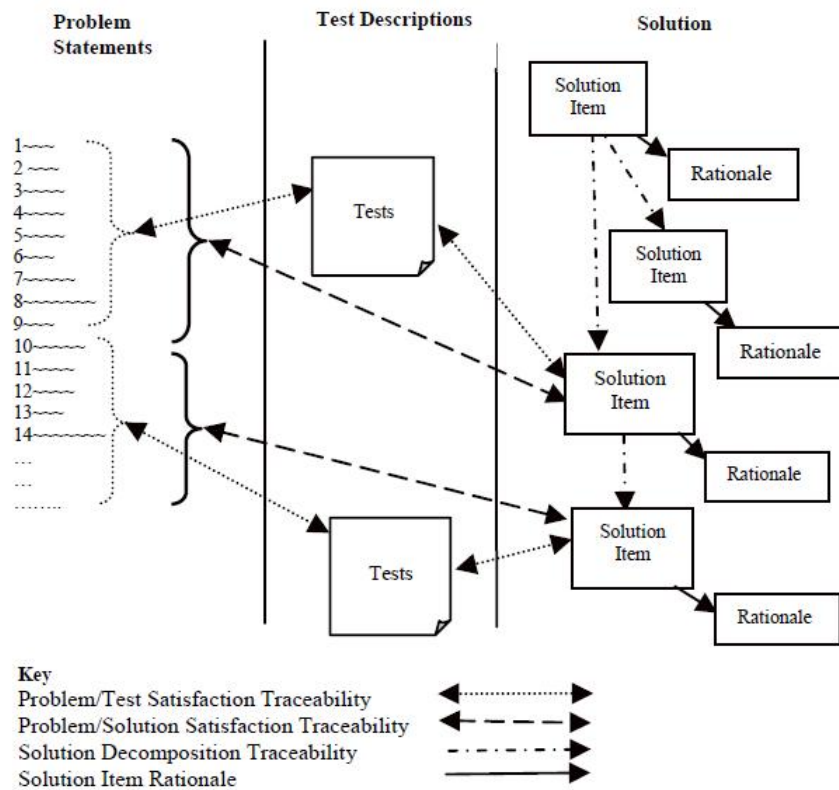


Figure 2.10: Traceability relating problem statements and solutions by (Arkley & Riddle, 2005)

2.4.4.4 Traceability for Understanding the System

One of the main uses of traceability is for understanding various artefacts in reference to the context in which they were created or in reference to other artefacts related to them (Spanoudakis & Zisman, 2005). Hence, traceability is of paramount importance in cases where artefacts are assessed or maintained by individuals other than the ones who created them. Furthermore, in development scenarios where many heterogeneous artefacts are developed, traceability can be used to understand the interdependencies between those artefacts (Aizenbud-Reshef *et al.*, 2006a)

Several approaches use traceability in order to relate source code to manual pages, requirement models and system documentation (Antoniol *et al.*, 2002; Maletic & Marcus, 2001; Marcus & Maletic, 2003). Moreover, rationalisation relationships have been proposed as a way for providing rationale about the form of requirements and system design artefacts (Pohl, 1996a; Pohl *et al.*, 1999; Ramesh & Dhar, 1992). (Arkley & Riddle, 2005; Ramesh & Edwards, 1993) also use trace links to relate artefact rationale with its related artefact.

(Sabetzadeh & Easterbrook, 2005) suggest the use of traceability information for mapping different viewpoints of a system to each other. (van den Berg *et al.*, 2006) trace relations to define crosscutting, i.e. the phenomenon which occurs when, in a mapping between source and target, a source element is scattered over many target elements and where in at least one of these target elements, one or more other source elements are tangled.

Another common use of trace links is in system reengineering where traceability information is used to relate components of a legacy system with components of a new system depending on their functionality (Ebner & Kaindl, 2002). Finally, in model transformation scenarios, trace links can be used for debugging the transformation.

2.4.4.5 Traceability for Reuse

Traceability can be also used to identify reusable artefacts at different levels of abstraction during the development process. After being identified, such artefacts can be reused through different scenarios. During the development of a new system, already implemented requirements could be identified and reused together with their related design and implementation (Constantopoulos *et al.*, 1995). This is a very frequent scenario in

product line software engineering, where a new variant of an existing system is created (Arkley & Riddle, 2005).

The abovementioned scenarios concern mainly traceability relations between artefacts at different abstraction levels. (Alexander, 2003) and (von Knethen *et al.*, 2002) propose the use of horizontal traceability for artefact reuse. Specifically, the former introduces a method to relate product requirements to existing use cases, while the latter proposes an approach for the reuse of coarse-grained requirement specifications expressed in structured text.

2.4.4.6 Traceability for Software Project Management

Another common use of traceability is for software project management purposes. (Gotel & Finkelstein, 1994) argue that trace links can be used for assessing the development process. This can be done since traces comprise a log of events occurred during the development of the system. This log could provide valuable information by being analysed using various metrics. Moreover traces can be used to monitor the status of requirements during development. That is requirements can be tracked through the various development phases such as implementation and testing (Arkley & Riddle, 2005; Ramesh & Edwards, 1993). From a project management point of view, traceability information can be used to rate requirements according to their risk or priority (Ramesh & Edwards, 1993). This can be achieved by using traceability relationships to trace requirements back to their goals. Finally, (Pohl, 1992) propose the use of traceability to identify and reuse best practices during software development.

2.5 Traceability in Practice

In the previous section, the relevant literature on traceability research has been presented. The purpose of this section is to discuss briefly the use of traceability in industrial settings. In section 2.5.1 the most well-known traceability tools are presented. In section 2.5.2, empirical studies about the use of traceability in industrial organisations are discussed, while in section 2.5.3 various limitations in current traceability practices are identified.

2.5.1 Traceability Tools

The main focus of this section is on traceability tools. There are three categories of tools. The first one consists of tools which support traceability as a secondary functionality. The second category consists of tools whose main functionality is traceability support. Finally, the third category consists of tool chains, in which traceability support plays an important role.

2.5.1.1 Tools with Traceability Support

There are many CASE tools which provide traceability support. The Requirements-Driven Design system (RDD-100) (Holagent, 2005) is a software product family product which is dedicated to requirements analysis, modeling, and design. The tool is built on an entity-relationship-attribute database and it supports functional flow modeling, data modeling, behavioural modeling, requirements analysis, report and document generation. Entity-Relationship-Attribute modeling is the basis through which requirements traceability is achieved. Requirements are represented as entities and relations define named bi-directional links to other requirements or other information stored within the tool.

Another example of a tool with traceability support is *Modelio* (Softteam, 2010), which is a modeling tool supporting dictionaries, requirements, business rules, and goals and report generation. Modelio provides full support of the UML standard as well as of the Business Process Modeling Notation (BPMN). The traceability support of the tool comes either in the form of traceability matrices or traceability models, as well as with a predefined set of traceability links.

Finally, the Software Concordance (SC) Editor (Nguyen & Munson, 2003) is a plug-gable architecture to support the integration of editors for different types of software artifacts. The SC environment is compatible with XML-based document editing environments since it supports the integration of editors for new document types whose internal representation is XML-compatible. Traceability links between different elements are represented using a formal hypertext model. This hypertext model can be defined as “a set of intellectual works and their inter- and intra-work relationships, represented by links, in combination with a user interface for viewing instances of these works and navigating from instance to instance across links” (Whitehead, 2000). A

work is an artefact that can be drawn from any medium, such as text or image. The links are mainly recorded manually by the user. The tool also provides facilities for the automatic recording of conformance links IR techniques (Maletic *et al.*, 2003), as well as support for the automatic handling of link evolution.

2.5.1.2 Specialised Traceability Tools

While in the previous section, we have discussed some examples of tools whose main functionality is not related to software traceability but they provide some limited traceability support, in this section we discuss some examples of tools which focus mainly on traceability support.

An example of such a tool is DOORS (IBM, 2010)- which is a requirement management tool. One of the main functionalities of this tool is to provide requirements traceability support. This is achieved by using hypermedia technologies. DOORS lets the engineer define attributes for the linked artefacts or even for the links. This attributes can define the purpose of links or their priority. For projects with process improvement goals, rules for link creation and direction can be defined, which helps users follow the process and prevent careless mistakes. With the addition of link information to history records. DOORS is a very intuitive and practical tool and this is how its popularity in the industry is explained. However, it is arguably not a very flexible tool, since it comes with a generic set of trace links and it does not distinguish between different kinds of links that can exist between different artefacts.

Another requirements management and traceability tool is RTM (Serena Software Inc., 2010). RTM is a requirements management tool implemented on top of an Oracle database. All requirement documents are stored in the database and each document has an *id*, while each paragraph has a number. Requirements can be refined and the trace link which relates the old and the new requirement is a *substitutes* link. Moreover, tests for the requirements can be defined and those tests can be linked back to the requirements using traceability information. Finally, the tool can perform impact analysis. When a requirement changes, the tool can determine which other artefacts are affected.

(Pohl, 1996a) proposes the PRO-ART tool, which is a requirements engineering environment. PRO-ART enables requirements pre-traceability by enabling the user to define and execute a requirements engineering process in a very fine-grained way. The

tool guides the user through the requirements elicitation and specification activities. As a result, traces between requirements are captured automatically by logging all the actions which take place in the tool and then by processing those logs. As a framework for structuring the trace information the tool adopts the Information Resource Dictionary Standard (IRDS) Standard (ISO/IEC, 1990).

While PRO-ART is focused on requirements traceability, the Traceability for Object-Oriented Requirements (TOOR) tool (Pineiro & Goguen, 1996) is more generic in the sense that TOOR is neither process-oriented nor restricted to requirements elicitation. TOOR's purpose is to enable traceability between requirements and other artefacts of the development process such as design modules or system documentation. The idea behind this tool is to specify the various traceable artefacts that can take part in a software development process, use TOOR to input these artefacts as they are created, and then trace requirements, making use of the relations among the current collection of artefacts. The aforementioned artefacts as well as traces between them are expressed in a declarative high-level object-oriented programming language called FOOPS. Moreover, TOOR provides a powerful mechanism for querying the various traceability information, which are generated and stored in the tool.

2.5.1.3 Tool chains

In this category of traceability tooling, the majority of the development activities takes place using different and often heterogeneous tools and traceability information is generated and managed at the level of the middleware, which is used for the integration of the various tools. An example of such an approach is the platform proposed by the OPHELIA project (Hapke *et al.*, 2004). Traceability in the OPHELIA platform takes place at the traceability layer (Smith *et al.*, 2003). Traceability is achieved by representing all artefacts of the software engineering process as CORBA objects and then allowing the engineer to define relationships between all CORBA objects in a dedicated tool which is called *Traceplough*.

In the approach proposed by the OPHELIA project, traceability information is generated and stored in the middleware which is used for tool integration. Hence, it is kept separate from the various related artefacts. Contrary to this, the approach proposed by

the GENESIS platform (Boldyreff *et al.*, 2002) keeps the related artefacts and the traceability information together. The GENESIS platform supports cooperation and communication among software engineers belonging to distributed development teams involved in modeling, controlling, and measuring software development and maintenance processes. It is made up of three main elements: a distributed workflow management system, a resource management system, and an artefact management system. The artefact management system is used to store and retrieve any item produced by any member of a software engineering team. Each stored artefact is accompanied by meta-data about its history and dependencies on other artefacts.

Another approach which attempts to integrate the traceability information in the different tools of the tool chain is the one proposed by (Sherba *et al.*, 2003). The tool they have developed is called *TraceM* and its conceptual framework is illustrated in Figure 2.11. The main elements of this framework are tool, artifact, relationship, and metadata. A tool is something that a stakeholder uses to perform a task such as a word processor or a UML diagramming tools. An artefact is produced by a tool. A relationship is a semantic association between artefacts, portions of artefacts, or relationships. Metadata allows a method engineer to describe the artifacts and relationships that are created and used during the project.

To conclude this brief presentation of the various traceability tools, we must acknowledge that a direct comparison between the various tools is very difficult for two main reasons. First, the various tools focus on different aspects of the development process such as the requirements elicitation phase or the whole development life-cycle. Therefore, the requirements for each tool are different. Second, while some of these tools are used widely in the industry (e.g. DOORS), others are only prototypes which are used in small industrial projects. Despite all the research efforts, tool integration is still a big issue in software traceability. (Aizenbud-Reshef *et al.*, 2006a) identify the lack of tool integration as one of the main challenges in traceability.

2.5.2 Empirical Studies

Various empirical studies have taken place in order to investigate the traceability practices in industrial organisations. These studies have indicated substantial differences

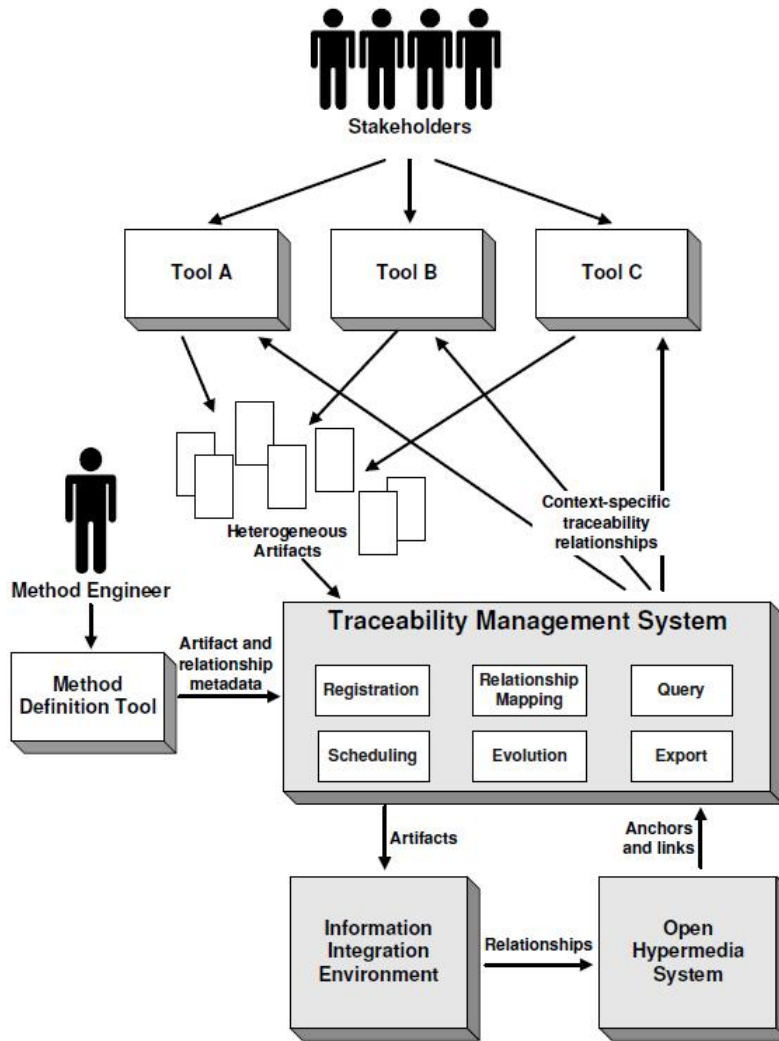


Figure 2.11: TraceM conceptual framework (Sherba *et al.*, 2003)

in the relevant practices, which depend on environmental, organisational and technical factors.

Following (Arkley & Riddle, 2005), one of the main factors that hinders the use of traceability in industrial projects is the ability to capture and maintain useful traceability information in a cost-effective manner. The main source of the problem according to (Arkley & Riddle, 2005) is the heterogeneity of the tools, which are used during software development as well as the lack of interoperability among those tools. Hence, traceability is considered to be an expensive activity especially for small or short-term projects.

The aforementioned cost of traceability is even higher if developers need to be trained in order to use the traceability tools and techniques. (Alexander, 2002) has found that most stakeholders are not familiar with tools and techniques to capture, manage and use traceability information. Moreover, (Gills, 2005) has found that for almost every IT project, organisations develop a new traceability metamodel. As a result this adds to the effort required for supporting traceability.

The accuracy and effectiveness of any analysis which uses trace links depends heavily on the quality of the trace links, namely the correctness and completeness of traceability information. However, these two properties can not be guaranteed for either manual or automatically generated links (Lindvall & Sandahl, 1998). As a result, organisations rely on the experience and knowledge of engineers in order to perform particular types of analysis instead of using captured traceability relationships.

Despite the aforementioned problems with the use of traceability in industrial settings, success stories can be found in the traceability literature. (Kirova *et al.*, 2008) describe their experiences in defining and deploying a traceability framework that meets the strategic goals of the host organization, its project needs and process improvement objectives. The derived benefits from their framework are reported in detail. A striking example of the effect of successful traceability practices on the software development process is the fact that using their framework reports and reviews that used to take one to two days to prepare, they can be obtained in seconds. A similar success story is reported by (Asuncion *et al.*, 2007), who present a successful end-to-end software traceability tool developed at Wonderware, a software development company.

The abovementioned success stories are not common though. Usually, trace links are not captured or maintained in a consistent and rigorous way. Moreover, they are almost

never consulted as support for software development activities. In the next section, we will briefly present the main factors, which affect the use of traceability in the industry.

2.5.3 Limitations

Although the advantages of traceability are well documented, traceability practice is not widely spread, as it can be seen by the various empirical studies. In this section, the limiting factors affecting traceability are briefly discussed.

(Winkler & von Pilgrim, 2009) provide a classification framework for categorizing the factors, which prevent the wide adoption of traceability practices in industry. This classification framework consists of four categories. These categories are:

- **Natural factors**, which are related to the imprecise and incomplete nature of traceability.
- **Technical factors**, which are related to the technical aspects of traceability.
- **Economic factors**, which are related to the difficulty to measure the Return on Investment (ROI) of traceability.
- **Social factors**, which have to do with how humans affect traceability practices.

We argue, that technical limitations to traceability arise mainly because of the other three limitations. If for example, traceability was complete and well-defined many of the recording technical limitations would not have existed. Hence, for this thesis, traceability is only limited because of natural, economic and social factors. In the following we will briefly discuss these three categories and how they can affect traceability practices.

2.5.3.1 Natural Factors

As was discussed in section 2.4, what traceability is, what kind of trace links should be captured, and how this information should be used is not universally agreed. There is no complete traceability scheme or metamodel and many researchers believe that such a metamodel can not exist (e.g. (Winkler & von Pilgrim, 2009)). This imprecise nature of traceability leads to many of the problems mentioned above. (Stone & Sawyer, 2005) argue that the information needed for capturing and maintaining traceability is

considered to be tacit. This makes the identification of the types of trace links needed for a project a difficult task.

Additionally, traceability practice is not only affected by the imprecise nature of traceability but also by the imprecise nature of other activities of the software development lifecycle. This is mainly the case during the initial phases of a project such as the requirement elicitation activities. It is very common the artefacts that result from these activities are expressed in natural language or in informal notations (Goguen, 1996). These artefacts are very often imprecise, ambiguous and incomplete. As a result, capturing and maintaining trace links from and to those artefacts is very difficult.

2.5.3.2 Economic Factors

These limiting factors to traceability practice adoption have mainly to do with the costs of traceability in relation to its benefits as it is perceived by the stakeholders of a project. Usually, it is considered by management that traceability is optional work for which insufficient resources are allocated (Wieringa, 1995). This is in accordance with the belief of many engineers that traceability recovery and maintenance is costing them a lot of effort (Flynn & Dorfman, 1990).

There is an inverse relationship between the cost of traceability and the degree of automation which can be achieved. That is, the higher the automation for capturing and maintaining traces, the lower the cost of traceability.

Another factor which affects the cost is the customisation needed for various traceability tools (Dmges & Pohl, 1998). Due to the project-specific nature of traceability, tools most of the time need some sort of customisation in order to be used for a particular project or in a particular organisation. As a consequence, the higher the effort for customising the relevant traceability tools or the interfaces among the tools of a tool chain, the higher the cost of traceability. As it is discussed in the previous sections, due to the nature of traceability it will never be possible to have a generic definition of trace types or a generic traceability tool which could fit exactly the needs of different projects. Consequently, traceability will always be a costly activity. What can be done though is that this cost can be minimised by providing ways to support project specific traceability. By minimizing cost of traceability and by acknowledging the big benefits of using traceability, traceability can be made attractive for project managers.

(Heindl & Biffel, 2005) propose a value-based requirements tracing process in order to address the issue of the high traceability cost. The goal of their approach is to identify traces based on prioritized requirements and thus to identify which traces are more important and valuable than others for the organisation. The authors have explored the effects of their approach using a case study. The findings of their case study are quite promising. Their approach was used in a real-life project setting and it was compared with ad hoc tracing and full tracing in terms of costs and benefits. One of the main results of the case study show that value-based traceability took around 35% less effort compared to full tracing. Moreover, risky requirements needed more detailed tracing and this was achieved with the proposed approach. One thing which remains unclear with this work is how the particular tools used for the purposes of tracing affect the results. Their findings were used to develop a tracing activity model (TAM) (Heindl & Biffel, 2008), as a framework for comparing different tracing strategies. This framework can be used to inform managers about the costs and benefits of alternative tracing strategies.

Following (Egyed, 2006), the cost and complexity of traceability can be dealt with by limiting trace analysis to some necessary minimum. Unfortunately, engineers rarely predict accurately which trace dependencies are more important than others. The author proposes a value-based approach to assist engineers into deciding which traces are needed, when they are needed and at what level of precision, completeness and correctness. This approach could reduce the cost of traceability by avoiding unnecessary trace recovery and maintenance. Cost of traceability can also be reduced by tailoring the precision, completeness and correctness of trace links depending on their intended usage. To investigate more about the trade-off between these three attributes and traceability cost, (Egyed, 2009) has conducted three case studies. The results of these case studies show that cost and effort of traceability can be reduced by reducing the granularity of the traceable artefacts. More precisely, they have found that *requirements-to-class-level* granularity provides higher value for money compared to *requirements-to-package-level* and *requirements-to-method-level granularity*. Consequently, they argue that reducing the granularity of the traceable artefacts reduces the benefit from using traceability information. Finally, the authors of this work argue that the essential attributes for a traceability approach are completeness and correctness, since they play an important role in follow up activities such as software maintenance.

2.5.3.3 Social Factors

It is widely accepted in the traceability literature, that traceability practices can never be fully automated (Egyed, 2006). Therefore, humans play an important role in capturing and maintaining traceability. Motivating the stakeholders who are responsible for traceability is of paramount importance for successful traceability practices. Lack of motivation is identified as a limiting factor to traceability by (Gotel & Finkelstein, 1994). According to this work, the individuals who are responsible for capturing traces are different from the individuals who use them. This results to low motivation on the side of the ones responsible for capturing the traces and consequently the quality of the captured traces is very low. Moreover, (Hayes & Dekhtyar, 2005) argue that human analysts do not trust the traces produced by automated traceability methods and many times they make the results of such methods worst. This can be the result of missing motivation or lack of understanding of traceability.

2.6 Concluding Remarks on Traceability

The focus on the previous sections was on traceability research, as well as on current traceability practices and limitations. Traceability is a very broad research area which covers the entire software development lifecycle. Since the first traceability tool (Pierce, 1978) in 1978, traceability research has gained increasing attention. Despite this, there are still many open issues regarding traceability. In Chapter 4, we will identify these issues and analyse the traceability research landscape by using a systematic, rigorous yet practical evolutionary method. This survey method will systematically capture the evolutionary interrelationships of current traceability approaches, thus providing a formal framework of understanding. This will be achieved by using the key characters from the existing approaches described in this chapter and by classifying those approaches on an evolutionary dimension based on their common descent.

2.7 Model Driven Engineering

The proposed approach to traceability in this thesis, which will be presented in the next sections, is based on the principles of Model Driven Engineering (MDE). Hence, in this

section, we will briefly discuss what MDE is and we will present the key topics relevant to this work.

2.7.1 A Basic Theory of Models

In the MDE approach to software engineering models are promoted to primary artefacts of the development process. Therefore, a thorough understanding of the basic principles underlying models and modeling core concepts is essential for comprehending the new paradigm in software engineering.

Models are of central importance in every aspect of human life. The capacity to build models is an innate ability, without which it would be impossible for humans to cope with everyday complexity. The unconscious process of model building leads to the reduction of this complexity by abstracting away unnecessary details of the human environment. In science, modeling is very crucial, since it is one of the determinants in theory development. In all engineering disciplines, models are used widely to predict the behavior of artefacts before they are actually built (Ludewig, 2003).

Models are of great significance in software engineering as well. Almost every process during the development of software implies some kind of modeling. Hence, software engineers and computer scientists spend a great deal of time building, testing and revising modeling languages, models and modeling tools. In short, models are one of the fundamental instruments of modern software engineering. Therefore, a thorough understanding of the basic principles underlying models and modeling core concepts is essential.

Giving a precise and universal definition of the term *model* is very difficult. The root of the term model is the Latin word *modulus*, which means measure, pattern, example to be followed. Since, models are applied in almost every facet of modern science and engineering, different and often conflicting definitions exist. From a philosophical point of view, a model is "a mental construction that, based on the reality, reproduces the main components and relationships of the analyzed segment of the reality (G. Kappel & Wimmer., 2006). In this definition, the term model is considered in the context of the empirical sciences. The functionality of such models is purely representational and the materialization of such mental constructs is not required. Models then act as a "simplification" of reality, abstracting away redundant details and emphasizing the features.

The relationship of the model and the reality is said to be *isomorphic*, a term borrowed from mathematics¹

In biology, "a model is a description of a system. A system is any collection of interrelated objects. An object is some elemental unit upon which observations can be made but whose internal structure either does not exist or is ignored; a description is a signal that can be decoded or interpreted by humans" (Haefner, 2003). Finally, quoting (Rothenberg, 1989):

Modeling, in the broadest sense, is the cost-effective use of something in place of something else for some cognitive purpose. It allows us to use something that is simpler, safer or cheaper than reality instead of reality for some purpose. A model represents reality for the given purpose; the model is an abstraction of reality in the sense that it cannot represent all aspects of reality. This allows us to deal with the world in a simplified manner, avoiding the complexity, danger and irreversibility of reality.

As it is evident from the definitions provided, there is no universal agreement as to what exactly a model is.

For the purposes of this work, we will use the definition provided by (Ludewig, 2003). Subsequently, "the particular strength of models is based on the idea of abstraction. A model is "*a representation of an original*". The idea of *abstraction*, which is used in this definition is actually what differentiates a model from a copy.

Various classifications of models exist. One such classification is that of (Seidewitz, 2003). He suggests that there are two classes of models- *descriptive* and *prescriptive* models. The former are models, which echo exactly the system under study. Such models are usually met in traditional scientific disciplines, such as physics and chemistry. In software modeling, these models are used to illustrate how a system works; for instance a Unified Modeling Language (UML) class model could be employed to describe the class structure of an object-oriented software system. The aforementioned UML model would be valid if the class structure was consistent with the descriptive model. The latter

¹"The word "isomorphism" applies when two complex structures can be mapped onto each other, in such a way that to each part of one structure there is a corresponding part in the other structure, where "corresponding" means that the two parts play similar roles in their respective structures." (Hofstadter, 1979)

class consists of models, which specify the system to be created (Ludewig, 2003). Such models are usually used in traditional engineering disciplines. Specifically, in software development models are employed to design software systems. So, the aforesaid UML model used to describe an object-oriented software system, can be also used to design the system. In the case of a prescriptive model, if the system deviates in anyway from the model, then it is the system, which is invalid and not the model (Seidewitz, 2003) .

A hierarchy of models used in software engineering is produced by (Cowling, 2005). This hierarchy is illustrated in Figure 2.12.

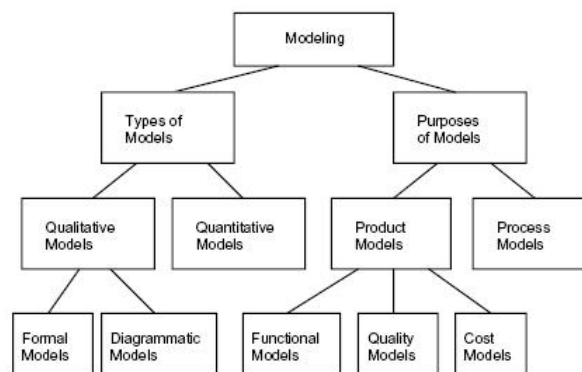


Figure 2.12: A hierarchical structure for modeling in software engineering (Cowling, 2005)

At the top level of his hierarchy there are two orthogonal aspects of modeling. The first of those aspects refers to the types of models that can be build and this can be subdivided into two more subcategories. One of these subcategories is concerned with qualitative or structural models, which relate to the description of the various properties of the entity modeled and the relationships among them. The other type of models at this level of the hierarchy is quantitative models, which model the measurable properties of the modeled entity. Further down, the qualitative models are subdivided into formal models and diagrammatic ones. The diagrammatic ones employ usually boxes and lines to model an entity and then visualize it (e.g. UML). The formal models use mathematical notations to model the world (e.g. the Z notation (Spivey, 1989)). Cowling identifies two different purposes for the models used in software engineering. The first one is concerned with the models used to describe software products, for example

models used to describe software architectures or requirements for a software system. The second purpose of the models used in software engineering is to describe software development processes. Finally, product models are subdivided into three different areas. Models, which describe the functionality of a system. Models, which describe the behavior of a system and finally models, which describe the cost of construction of a system.

Another criterion, which can be used to classify models is the degree of formalism used for the model specification and verification. Formal models have unambiguous and precise semantics. This is accomplished by the utilization of mathematical notations such as Z (Woodcock, 1996) and B-method (Wordsworth, 1996). Even though formal methods add rigor in the process of defining models, software developers seem unwilling to use them. This is mainly due to several issues related to formal methods such as the inability of mainstream programmers to apply them (Bowen, 1994) or their limited scalability (Sommerville, 2004). Conversely, semi-formal models employ informal methods such as natural language, in to define their semantics. By doing so, greater flexibility is achieved, since different interpretations in different contexts are allowed. Unlike formal methods, semi-formal methods are used widely in the software industry. Examples of such semi-formal modeling technologies are MOF (Object Management Group, 2006a) and EMF (Eclipse Foundation, 2010a).

Following (Ludewig, 2003), who quotes a German text written by (Stachowiak, 1973), a model must meet three criteria so as to differentiate itself from any other artefact. The first one is the *mapping criterion* according to which a model should be mapped to a phenomenon or object. The second one is called *reduction criterion*. As indicated by this criterion, not all properties of the system under study should be mapped to the model describing it. The fewer the amount of properties mapped, the higher the level of abstraction of the model. Finally, the last criterion is the *pragmatic criterion*, which suggests that the original can be substituted by the model for some particular purpose. If and only if all these criteria are met, an artefact can be considered to be a model.

Before concluding this brief introduction to models, two more modeling concepts should be introduced. The first one is the concept of a modeling language. According to (Seidewitz, 2003), a modeling language supports the representation of statements in models of particular systems under study. For the time being, the most prominent

modeling language in the field of software modeling is UML. The second concept is that of the interpretation of a model. By interpreting a model, the model elements are mapped to the elements of the system modeled. By doing so, the truth value of the statements in the model can be determined.

The above discussion summarizes the fundamental key terms and concepts behind models and modeling in general, using examples from the domain of software engineering, where applicable. Understanding the basics of model theory is imperative, since models play a crucial role in software development practice. In the following, we will briefly present the basics of metamodeling and MDE.

2.7.2 Metamodeling

The old modelling practices of software engineering used to employ models only as documentation for the produced software. This is in contrast to the modern practices of MDE, where models are used as formal input/output of computerised tools (Bezivin, 2005). For this reason, models and modeling languages should be defined in a precise and unambiguous manner. This is accomplished by the use of metamodels. Quoting (Seidewitz, 2003)

A metamodel is a specification model for a class of systems under study (SUS), where each SUS in the class is itself a valid model expressed in a certain modeling language. That is, a metamodel makes statements about what can be expressed in the valid models of a certain modeling language.

More simply put, a metamodel is the model of a modeling language. It consists of a collection of the construct and rules needed to define a semantic model. The term *meta* is just used to stress the fact that a metamodel describes a modeling language at a higher level of abstraction. The activity of constructing metamodels is called *metamodeling*. The basic uses of metamodels are language definition, domain-specific modeling and model interchange (Gitzel & Korthaus, 2004).

Even though a metamodel is defined as a model, it is still differentiated from models by two characteristics. The first one is that fact that a metamodels must belong to a metamodel architecture. That is, it must belong to a framework, within which attributes of a metamodel can be realized. This framework defines how metamodeling levels are

related to one another (Atkinson & Khne, 2001). The concept of a metamodel architecture is closely related to a metamodel hierarchy. Following (Gitzel & Hildenbrand, 2005), a metamodel hierarchy is defined as “a tree of models connected by instanceOf relationships”. In this hierarchy, there is a reoot model and all the models, which have the same distance form this root model belong to the same layer or level of the hierarchy.

The second characteristic, which distinguishes metamodels from models, is the fact that they must have the expressiveness to capture all the essential features and properites of a modeled language, i.e. its abstract syntax.

The above discussion summarises the conceptual framework of metamodeling. Meta-modeling plays a crucial role in MDE, since it provides the necessary facility to define semantically rich languages. This is imperative in the context of MDE, since languages with exressional power can precisely capture various aspects of a problem domain.

2.7.3 MDE: Concepts and Practices

MDE provides a conceptual framework for the development of software intensive systems. (Kent, 2003), states that MDE combines methods, frameworks, processes, tools and modeling languages in order to produce, efficiently and rapidly, quality software. In the crux of MDE is the fact that all the artefacts produced during the software development process are models, which are expressed in modeling languages. These models are subjected to subsequent transformations until they reach a final state, where executable code is produced. The MDE approach to software development differs from all the other traditional development methods in the way artefacts are produced in each development phase as well as in how these artefacts are processed. In MDE, as mentioned earlier, all the artefacts are models, which are expressed in a well defined modeling language. This enables the developers to manipulate and to transform all the produced models using specialized computer-based tools (Alanen & Truscan, 2003). Such tools maximize the gains from having models and minimize the effort and time needed to maintain them (Kent, 2003).

(Alanen & Truscan, 2003) describe the MDE method using a 4-layer hierarchy. Every layer contains functions not found in any other layer. These functions are used by the next layer in the hierarchy.

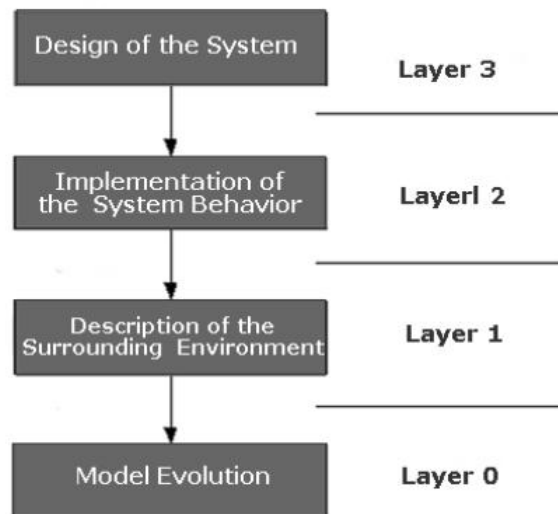


Figure 2.13: Traceability Definitions

In layer three of the aforementioned hierarchy, the design of the system takes place following a systematic approach. Firstly, the functional specification of the system occurs until reaching a desired level of detail. Functional requirements are defined and then these requirements are specified in a platform independent approach. At this point, developers are only concerned about the functionality, which the system must deliver, and the platform-specific details are not taken into consideration until later in the process. Subsequently, the initial broad specification is transformed into a domain dependent specification using domain based knowledge. By doing so, various generic reusable components can be specified for the particular problem domain. The resulting domain specific specification and the operations it will provide, will be used later to bridge the functional specification with the platform dependent specification. The last stage of this conceptual layer is to transform the domain dependent specification into platform dependent. Following this design methodology, different target platforms can be chosen for implementation. To do so, designers have to consider the resource needs of the domain operations derived in the previous stage, and then they have to ascertain which platforms are capable of offering these resources. Finally, the domain specification is directly mapped onto the chosen platforms. The end result of this process is a functional specification of the system, which is both domain and platform dependent (Alanen &

Truscan, 2003).

In layer two of the hierarchy, the behaviour of the system is implemented using mainly model transformations and model manipulation. In this level MDE scripts, expressed in the available modeling languages, are used extensively. These scripts are small applications able to query or transform a target model. Three types of scripts are required in the MDE process, which can be categorized according to the functionality they offer. The first type is query scripts. These scripts are usually employed to gather information from models, but they do not alter the model under examination in any way. Other functions of query scripts are to define model constraints, design guidelines and diagnostic software metrics. The second type of scripts required in an MDE process is model transformation scripts. These scripts can be either update transformation or mapping transformation scripts according to the scope of their effect on the target model. An update transformation script alters specific elements of a model. It comprises several queries, which select particular elements from the model under consideration, and then a delete, update, or create operation, which modify the selected elements. Thus, the source model and the target model coincide. Update transformations are usually employed when only a small part of the model needs modification. A special case of update transformations, where the whole model is being modified, is called model refinement or refactoring. Following Beck, a refactoring is “*a change to the system that leaves its behavior unchanged, but enhances some non-functional quality—simplicity, flexibility, understandability, ...*” (Beck, 1999).

The second type of transformation script is the mapping transformation. Contrary to update transformation, a mapping transformation transmutes elements of the source model to elements in one or more target models. Depending on whether the source and target models belong to the same formalism, mapping transformation scripts can be further categorized to model translation and language translation scripts. The former can be used to define mappings between models in the same language, while the latter can be used to define mappings between models in different languages (Kent, 2003). Another classification of a mapping transformation can be done according to the level of abstraction of the source and target models. A horizontal transformation is a transformation which takes place among models, which reside on the same level of abstraction. On the other hand, a vertical transformation is a transformation, which takes place among models, which reside on different levels of abstraction (Mens *et al.*, 2005). The main

difference between the mapping transformation approach and the update transformation approach is that when a mapping transformation is performed a new model is produced, while the source one remains intact (Gerber A., 2002).

One of the fundamental aims of model transformations is to generate artefacts throughout the software development process. Those artefacts are produced using specialized computer-based tools and can range from documentation or alternate specifications to code. The aforementioned tools must possess three characteristics. To begin with, such tools must comply with the syntax of the target language. Then, they have to comply with the semantics of the modeling language and to end with, they have to support traceability and reverse engineering. The first characteristic guarantees the syntactic correctness of the derived artefacts. Such a verification process is trivial according to (Alanen & Truscan, 2003), since target languages comply to a well defined grammar (e.g. BNF). The second characteristic guarantees semantic equivalence between the target language and the the source model. Lastly, traceability and reverse engineering should be features of the aforesaid computer-based tools, so that particular parts of the target language could be mapped back to the original source model.

In the last two layers of the MDE hierarchy under consideration, the surrounding environment of the MDE is described. This includes the interoperable tools used in MDE, the facilities which are provided by those tools, available languages and model manipulation rules. One of the indispensable advantages of the MDE techniques stems from the fact that various phases of the software development process are automated by the use of specialized tools. It is crucial for the success of an MDE approach that the abovementioned tools are able to communicate with each other, since MDE is utilized in such a vast domain of applications, which encompasses many diverse variables, that it is impossible for a single tool to cover all of them. A prerequisite for the communication of these tools is the existence of a standard interchange format, with which all the tools will comply (Stevens, 2003). An essential function, which should be provided by the CASE tools, is a versioning system. As described earlier in this paper, the MDE approach comprises successive transformations of models until executable code is generated. Nonetheless, the intermediate versions of models should not be discarded. On the contrary, they should be stored and there should be a way the various tools to be able to discern among the various versions of a model. Such a functionality can be very use-

ful, since transactional support for the various transformations can be realized (Alanen & Truscan, 2003).

2.8 Chapter Summary

This chapter has presented the core principles of software traceability and identified the core traceability activities. For each activity, the state-of-the art approaches have been identified and a discussion on their advantages and shortcomings has been provided. Following that, a discussion on traceability practices has been presented. Based on existing empirical surveys, limitations of current traceability practices have been discussed. At the second part of this survey, the principles of Model Driven Engineering have been identified. These principles form the basis on which this work is founded. In the next chapter we will identify the open research issues related to traceability based on the conducted literature survey.

Chapter 3

Analysis of Software Traceability & Hypothesis

In Chapter 2, a detailed review of the software traceability field was conducted. In this review, the different approaches to traceability are presented and open issues are identified. In this chapter, these findings are summarised and further analysed by using a classification approach borrowed from Biology called *phenetics* or *numerical taxonomy*. By using phenetics, we provide a formal framework for understanding the different areas of traceability research by using different clustering techniques to classify the various approaches. Based on the phenetic analysis, we will establish the research hypothesis as well as the research objectives of this work. Finally, in the last section of this chapter the research methodology will be outlined.

3.1 Introduction

A classification, as usually understood, allocates entities to initially undefined classes or categories so that entities in a class are in some sense close to one another (Cormack, 1971). Following (Hempel, 1952), classification is a process which is basic to all scientific and engineering domains, since it generates the concepts upon which a discipline can begin to build an understanding of the phenomena within its domain. Moreover, (Cormack, 1971) acknowledges that the most important step in conducting any form of scientific study involves the ordering and classification of objects under study. In

Cormack's lecture to the Royal Statistical Society, the benefits of a classification were summarised:

... the information about the entities is represented in such a way that it will suggest fruitful hypotheses which cannot be true or false, probable or improbable, only profitable or unprofitable.

Numerous classifications have been created to study software traceability approaches (e.g. (Dahlstedt & Persson, 2003; Pinheiro, 2004; Spanoudakis & Zisman, 2005; von Knethen & Paech, 2002; Wieringa, 1995; Winkler & von Pilgrim, 2009)), but to our knowledge these classifications are informal in nature and none of them refers to, or applies, any concepts or techniques from the science of classification. Thus, the purpose of this section is to produce a well defined theoretical classification of software traceability approaches. This classification will be then used to establish the research hypothesis for this thesis. To this end, *Numerical Taxonomy* or *Phenetics* (Sneath & Sokal, 1963), a formal and scientific approach to building classifications, will be used.

In biological sciences, there are two main types of classification systems, phenetic and phylogenetic. Phenetic systems, also known as *taxometrics*, is the numerical evaluation of the similarity between taxonomic units and the ordering of those units into categories (taxa) on the basis of their affinities (Sneath & Sokal, 1973). The primary aim of the approach is repeatability and objectivity. Phenetic techniques include various forms of clustering and ordination, so that the variation displayed by entities is reduced to a manageable level. In practice this means measuring dozens of variables, and then presenting them as two or three dimensional graphs.

On the other hand, phylogenetic systems group entities of interest purely on the evolutionary interrelationships of the various taxonomic units. This is achieved by using morphological data matrices. The term phylogenetics is of Greek origin from the terms *phyle*/*phylon*, which means "race" and *genetikos*, which means "relative to birth". Phylogenetic systems are also referred to as *Cladistic classification* in honour of the German entomologist Willi Hennig, who proposed the theory of Phylogenetic Systematic Principle (PSP) (Ridley, 1993).

A classification problem is usually referred to as the λ -problem (Leseure, 2000) or the ϕ -problem (McCarthy & Tsinopoulos, 2003). The λ -problem for this thesis aims to group the various traceability approaches according to their similarity and then identify

knowledge gaps in the existing body of work. Since this investigation does not focus on the evolutionary relationships between the various approaches but only on their overall similarity, the phenetic method of classification seems the most appropriate. This is due to the fact that this method considers the overall similarity of the taxonomic units and it does not distinguish between plesiomorphies - traits that are inherited from an ancestor - and apomorphies - traits that evolved anew in one or several lineages. In summary, the aim of a phenetic classification of traceability approaches is to yield taxonomic groups which bring together approaches with the highest proportion of similar attributes. Such a classification could provide a system for conducting, documenting and coordinating comparative studies of those approaches. In the next section, the basics of phenetics will be briefly presented.

3.2 Phenetics

Phenetics establishes classifications of entities based on their similarities. It utilizes many equally weighted attributes of the taxonomic units and employs clustering algorithms to yield objective groupings. It is out of the scope of this work to describe in detail how phenetics works. However, a brief description of the phenetic process will be provided in this section.

The construction of a system of classification by numerical phenetic methods involves six operation steps (Sneath & Sokal, 1973):

1. Select and assemble the units to be classified.
2. Select the characters and construct the data matrix.
3. Estimate the similarity among the units, based on their characters.
4. Formulate the taxonomic groups based on degrees of resemblance among the units.
5. Validate statistically the results in order to find the best solution.
6. Interpret the results of the clustering.

Select and assemble specimens: The first step in the phenetic process is to select and assemble the specimens, i.e. the entities under study. In the jargon of numerical taxonomy, a specimen is referred to as an *Operational Taxonomic Unit (OTU)*. For the purposes of this study, the OTUs are the traceability approaches presented in section 2. For numerical analysis, the various OTUs are selected non-randomly. For example, several approaches have been excluded from this study on the grounds that their documentation is limited. Due to the non-random nature of the sample, this approach should be used only as a descriptive method for measuring the similarity in the sample. Any conclusions ascribed to a larger population from which the sample was drawn (in this case all the existing traceability approaches) must be based on analogy and not on inferential statistics. The nonrandom sample of entities prevents us from testing statistical hypotheses about the results of analysis. Hence, we must rely on informed judgment to assess the risk of extrapolating our findings to the population from which the sample was drawn. However, following (Romesburg, 1984), if a sample seems typical of a larger population of entities and if the sample size is large, the probability of incorrect extrapolation is reduced. For the purposes of this study, although not all the existing traceability approaches are included in the study, most of the approaches found in the literature are included. This means that the sample size corresponds to a high proportion of the entire population. Based on that, the results from the phenetic analysis can be used for making assumptions about the entire population.

Forty-five traceability approaches have been chosen based on the availability of sufficient information for each approach. From a sampling theory point of view it is an intentional non-probabilistic sample. These OTUs are listed in Table 3.1. If the approach is supported by a tool, the name of the tool is provided in the parentheses. Since an in depth literature review took place in chapter 2, a detailed discussion of the OTUs will not be provided here.

Select the characters and construct the data matrix: Every one of the OTUs listed in Table 3.1 possesses a set of characters, i.e. a set of observable features or attributes. The terms *character* and *attribute* will be used interchangeably in this work. Characters play a fundamental role in a resulting classification since specimens are grouped based on their characters. Quoting (Davis & Heywood, 1963), “although it is the organisms which are classified, it is their characters which provide the evidence used in

#	Source	#	Source	#	Source
1	(Amar <i>et al.</i> , 2008) (Etrace)	16	(Lin <i>et al.</i> , 2006) (ArchTrace)	31	(Sharif & Maletic, 2007)
2	(Antoniol <i>et al.</i> , 2002)	17	(Lucia <i>et al.</i> , 2007) (ADAMS ReTrace)	32	(Sherba <i>et al.</i> , 2003) (TraceM)
3	(Cleland-Huang <i>et al.</i> , 2003)	18	(Lucia <i>et al.</i> , 2008) (ADAMS ReTrace)	33	(Sousa <i>et al.</i> , 2008)
4	(Costa & da Silva, 2007) (RT-MDD)	19	(Mäder, 2008) (TraceMaintainer)	34	(Spanoudakis <i>et al.</i> , 2003)
5	(Egyed & Grunbacher, 2002) (Trace Analyzer)	20	(Maletic <i>et al.</i> , 2003)	35	(Spanoudakis <i>et al.</i> , 2004)
6	(Fabro <i>et al.</i> , 2005) (Atlas Model Weaver)	21	(Maletic <i>et al.</i> , 2005)	36	(Spanoudakis & Zisman, 2005)
7	(Falleri <i>et al.</i> , 2006) (KERMETA)	22	(Marcus & Maletic, 2003)	37	(Vanhooff <i>et al.</i> , 2007a) (UniTi)
8	(Grammel & Voigt, 2009) (Trace-DSL)	23	(Murta <i>et al.</i> , 2006) (Poirot TraceMaker)	38	(von Knethen & Grund, 2003) (QuaTrace)
9	(Grechanik <i>et al.</i> , 2007) (LeanArt)	24	(Natt och Dag <i>et al.</i> , 2005) (ReqSimile)	39	(Walderhaug <i>et al.</i> , 2006)
10	(Hayes <i>et al.</i> , 2003) (RETRO)	25	(Olsen & Oldevik, 2007) (MOFScript)	40	(Wenzel <i>et al.</i> , 2007)
11	(Hayes <i>et al.</i> , 2004) (RETRO)	26	(Pinheiro & Goguen, 1996) (TOOR)	41	(Bayerand & Widen, 2001)
12	(Hayes <i>et al.</i> , 2006) (RETRO)	27	(Pohl, 1996a) (Pro-Art)	42	(Ying <i>et al.</i> , 2004)
13	(Jirapanthong & Zisman, 2007) (Xtraque)	28	(Ramesh & Jarke, 2001)	43	(Zimmermann <i>et al.</i> , 2004) (ROSE)
14	(Jouault, 2005) (ATL)	29	(Rose <i>et al.</i> , 2008) (EGL)	44	(Zisman <i>et al.</i> , 2003)
15	(Kolovos & Paige, 2010) (EPSILON)	30	(Schwarz <i>et al.</i> , 2009) (Graph-based Traceability)	45	(Zou <i>et al.</i> , 2006)

Table 3.1: Operational Taxonomic Units for the phenetic analysis

classification”. In selecting the taxonomic characters, three types of characters should be ignored:

- Characters that do not reflect the inherent nature of the OTU (e.g. if an approach is supported by a tool or not).
- Characters, which do not vary in the group (e.g. all the approaches under consideration deal with traceability).

- Characters, which are invariant, i.e. add no information (e.g. the name of the creator of an approach).

The basic unit of information for a phenetic study is called a *unit character*. The unit character is defined as a “taxonomic character of two or more states, which within the study at hand cannot be subdivided logically” (Sneath & Sokal, 1963). The selection of characters for use in a phenetic study is of paramount importance, since these characters form the basis of the classification. In computing the similarity between taxonomic units, all taxonomic characters are treated as of equal value and importance. Due to this all character have the equal weight during the classification process.

The requisite minimum number of characters for a phenetic study is not known. (Sneath & Sokal, 1973) recommend not less than sixty characters. However, they admit that a requirement of such a number is arbitrary and cannot be justified. Hence, it seems sensible to use as many characters and of as diverse nature as possible, in order to obtain good representations of the entities involved.

Determining the characters is a two-step process - the search for characters and the selection of characters. The former step is an idiosyncratic process, while the latter includes decisions such as rejecting a character as irrelevant or as introducing noise to the data (McCarthy & Tsinopoulos, 2003). The process of determining the relevant characters for a classification problem is also called character *mining*. The mined characters for this thesis are listed in table 3.2. In Appendix A, each of the mined characters is explained briefly.

The data set for the classification problem at hand consists of qualitative attributes measured on nominal scale. Some of the attributes are binary, i.e. they have only two states, while others are multistate. Since all the attributes are qualitative, they do not need to be *standardised*. To code the multistate attributes, we have used *dummy* binary attributes. For example, a multistate attribute is the *automation of the identification activity* of a traceability approach. The states of this attribute are *manual*, *semi-automatic*, *automatic*. To represent this multistate attribute, we have created three different binary *dummy* variables, namely the “*manual identification of links*” attribute, the “*semi-automatic identification of links*” attribute and the the “*automatic identification of links*” attribute. These three attributes have only two states - the “*present*” and the “*absent*” states. The coding of the attributes took place based on the information

#	Character	#	Character	#	Character
1	Manual Identification of Links	14	Implicit identification with transformations	27	Text-to-text traceability
2	Semi-automatic Identification of Links	15	Explicit identification with transformations	28	Model-to-text traceability
3	Automatic Identification of Links	16	Guidance for traceability usage	29	Model-to-model Traceability
4	Identification with IR techniques - VSM	17	Inter-artefact storage of traceability information	30	Artefact-specific support
5	Identification with IR techniques - LSI	18	Intra-artefact storage of traceability information	31	Artefact-agnostic support
6	Identification with IR techniques - PM	19	Case-specific traceability metamodel - extensible	32	State-based maintenance
7	Identification with indirect rules	20	Case-specific traceability metamodel - non-extensible	33	Event-driven maintenance
8	Identification with direct rules	21	Generic traceability metamodel	34	Automatic traceability maintenance
9	Identification with program analysis	22	Representation with hyperlinks	35	Manual traceability maintenance
10	Identification with run-time monitoring	23	Representation with graphs	36	Semi-automatic traceability maintenance
11	Identification augmented with A.I. techniques	24	Representation with inline tags	37	Dependency viewpoint support
12	Identification augmented with visualisation techniques	25	Representation with matrices	38	Generative viewpoint support
13	Identification with history analysis	26	Semantically-rich link semantics		

Table 3.2: Attribute table

provided in Chapter 2. While the coding process is subject to error, the validity of the process is guaranteed by the fact that numerical taxonomy relies on a large number of characters. Therefore, if the number of errors is kept within reasonable bounds, the correctly coded characters should largely determine the relationships generated by the cluster analysis. Table B.1 in Appendix B lists the different states for the 38 attributes, while table B.2 shows the states possessed by each OTU.

The estimation of resemblance: Resemblance among the OTUs is estimated by calculating a coefficient of similarity or dissimilarity between all pairs of the taxonomic units over all attributes of the data matrix. Many such coefficients have been proposed

in the literature. A thorough review of the various similarity/dissimilarity metrics is provided in (Choi *et al.*, 2010). The difference between a dissimilarity and a similarity coefficient is merely a difference in the way the scale runs. The smaller the value of a dissimilarity coefficient, the more similar the two OTUs under consideration are. On the other hand, the larger the value of a similarity coefficient is the more similar the the two OTUs under consideration are. The values for every pair of OTUs are stored in a resemblance matrix.

For the purposes of this work we have chosen the following coefficients:

1. Kulczynski similarity distance
2. Dice similarity distance
3. Hamming dissimilarity distance
4. Jaccard similarity distance
5. Sokal-Sneath dissimilarity distance

The above coefficients assume that the data are binary. The choice of those coefficients was made based on the fact that they all ignore the “0-0” matches. In any matching problem with binary data, there are four different types of matches. The first type of match is the so-called “1-1” match, that is an attribute is present in both OTUs. The other two types are the “1-0” match and the “0-1” match, in which an attribute is present in only one of the OTUs. Finally, the “0-0” match type is when an attribute is absent from both OTUs. The above formulas ignore the “0-0” matches, assuming that such matches do not contribute to the similarity of two OTUs. This is a logical assumption for this study since many OTUs are lacking many of the attributes. Hence, these attributes do not vary significantly across the set of OTUs. As a result, they can not contribute to the discrimination between classes of objects (Romesburg, 1984).

In Table 3.3, the formulas of the coefficients are illustrated. In these formulas, a stands for the “1-1” match, while b and c stand for the “1-0” match and the “0-1” match respectively.

The dissimilarity matrices for the five coefficients are shown in Appendix C. Each entry represents the dissimilarity between a pair of approaches. For example the (1,2) entry of the matrix depicts the dissimilarity between approach number 1 (Amar *et al.*,

Coefficient	Formula
Kulczynski distance	$\frac{a/2(2a + b + c)}{(a + b)(a + c)}$
Hamming distance	$b + c$
Dice dissimilarity	$\frac{2a}{2a + b + c}$
Jaccard distance	$\frac{a}{a + b + c}$
Sokal-Sneath dissimilarity	$\frac{a}{a + 2b + 2c}$

Table 3.3: Similarity/Dissimilarity Coefficients Formulas

2008) and approach number 2 (Antoniol *et al.*, 2002). The resemblance matrix is a hollow symmetric matrix. It is hollow since the values on the main diagonal correspond to pairs of the same approach, whose distance is zero. The next step is the recognition of patterns in the dissimilarity matrix. This is the area of numerical taxonomy which utilises clustering techniques.

Formulate the taxonomic groups: In this step, the resemblance matrix is transformed into a dendrogram. This is achieved by using a clustering method. A clustering method is a series of steps that incrementally reduce the size of the resemblance matrix, generating the dendrogram from the values of the matrix. Once a value is used for generating a branch of the dendrogram, then this value is deleted by the resemblance matrix. At the end, the matrix disappears completely. The dendrogram consists of various clusters. A cluster is a set of one or more objects that they are similar (Romesburg, 1984).

Following (Romesburg, 1984), there are four main clustering methods:

1. Unweighted pair-group method using arithmetic averages (UPGMA)
2. Ward's method

3. Single-linkage method

4. Complete-linkage method

A thorough discussion of the various clustering methods is out of the scope of this work. At this point though, it should be noted that the Ward method is most appropriate for quantitative data and not for binary data. Hence the other three will be used for the purposes of this analysis. These methods can possibly produce very different classifications, resulting in fifteen ($5 \text{ resemblance coefficients} \times 3 \text{ clustering methods}$) different classifications. Since the result of the analysis so far gives more than one classifications, we need a method for deciding which one of those classifications is the best. This is achieved in the next step of the process.

Statistically validate the results: Since the dendrogram used to show the results of a cluster analysis is a two-dimensional representation of a multi-dimensional structure, some distortion of the relationships in the similarity matrix on which it is based is inevitable. That is, it is often the case that the clustering method distorts the information in the resemblance matrix in order to produce the dendrogram.

The cophenetic correlation coefficient (CCC) can measure how well a dendrogram and a resemblance matrix fit each other (Romesburg, 1984). The computation of the CCC is a two-step process. First, a matrix of cophenetic values is obtained from the dendrogram by finding the similarity level that links each pair of OTUs. The cross-product correlation coefficient is then computed between the cophenetic matrix and the resemblance matrix. This is the CCC. A value of one represents complete agreement between the two matrices. The higher the CCC is, the closer the dendrogram fits the resemblance matrix.

Table 3.4: CCC for the Different Dendrograms

	Kulczynski	Dice	Hamming	Jaccard	Sokal-Sneath
UPGMA	0.861	0.799	0.754	0.832	0.881
Single-linkage	0.744	0.666	0.559	0.740	0.802
Complete-linkage	0.786	0.761	0.737	0.810	0.848

Table 3.4 shows the CCC for the different combinations of clustering methods and resemblance coefficients. For five combinations, the CCC indicates high concordance since it is larger than 0.8 (Romesburg, 1984). To select the classification with the smallest error, the classification with the highest CCC must be chosen. This classification is the one provided by using the Sokal-Sneath dissimilarity coefficient and the UPGMA clustering method. The dendrogram which corresponds to this combination is illustrated in Figure 3.1. Via inspection, we “cut” the tree at $distance=0.8$.

All computations in this study were conducted using the *Scipy-cluster* package (Eads, 2008), which is a Python library for agglomerative clustering. The last step of the phenetic process is the interpretation of the dendrogram, which is presented in the following section.

The main limitations of this study are of two kinds: limitations with the numerical taxonomy itself and limitations with the construction of the data matrix. A detailed discussion of the limitations of numerical taxonomy is provided by (Queiroz & Good, 1997). On the other hand, the limitations imposed by the construction of the data matrix have to do partly with the subjectivity of selecting attributes as well as with the information available to the researcher for the coding of the attributes. Attribute mining is a subjective task based on the researcher’s intuition and knowledge of the domain. To compensate this subjectivity, as many as possible attributes were chosen for the conducted analysis. Moreover, the quality of the data matrix was influenced by the amount and quality of the information provided by the documentation of each approach. Coding some of the attributes was a challenge since not enough information was available for some of the approaches. In these cases, interpretations and assumptions were made based on the intuition of the analyst.

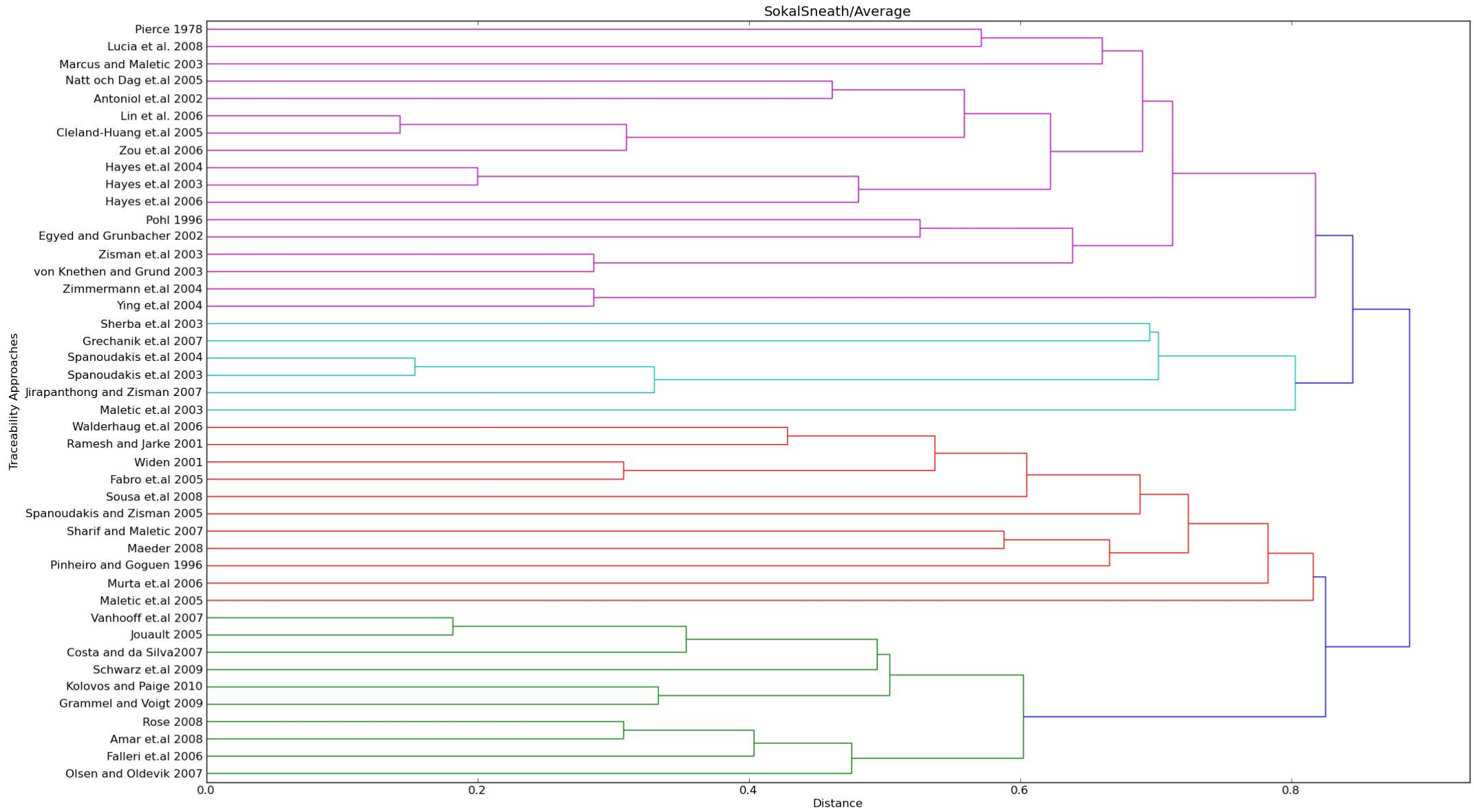


Figure 3.1: Dendrogram of Traceability Approaches

3.3 Discussion

In the previous section the phenetic process was followed in order to construct a classification of different traceability approaches. The result of this process is captured in the form of a dendrogram, which is illustrated in Figure 3.1. The purpose of this section is twofold. First, open issues in the area of traceability will be identified based on the literature survey conducted in chapter 2 and on the data matrix illustrated in Table B.2. Second, an interpretation of the dendrogram produced in the previous section will take place.

3.3.1 Analysis of the Data Matrix

By studying Table B.2 as well as the survey conducted in chapter 2, the following observations can be made about the traceability approaches under study:

Observation # 1 - Coverage of traceability activities: In chapter 2 four different activities related to traceability have been identified, namely traceability representation, traceability identification, traceability maintenance and traceability usage. The only two traceability approaches which cover all four aspects of traceability is (Schwarz *et al.*, 2009) and (Grammel & Voigt, 2009). The rest of the approaches are concerned with one or two specific aspects of traceability only. As a result, they are defined in a rather isolated way, using different, not necessarily combinable techniques and technologies. This consequently makes their integration into a comprehensive traceability environment a challenge.

Observation # 2 - Support for Dependency and Generative Viewpoints: In section 2.4.1.1, the two viewpoints associated with traceability are presented. In the Dependency viewpoint, the focus of traceability is on creating and maintaining traceability relationships between existing artefacts. On the other hand, in the Generative viewpoint the focus on traceability is on creating and maintaining traceability relationships as a byproduct of transformations between artefacts. A comprehensive traceability approach should be able to support both of those viewpoints. The only approach of the approaches under study that is considering both of the aforementioned viewpoints is the RT-MDD (Costa & da Silva, 2007). The rest of the traceability approaches focus on one viewpoint. Most of the approaches, which originate from the MDE community (e.g. (Falleri *et al.*, 2006; Jouault, 2005; Olsen & Oldevik, 2007)), focus on the Generative viewpoint, while the rest focus on the Dependency viewpoint.

Observation # 3 - Link Semantics and Automation: The need to distinguish between different types of trace links with specific semantics is widely accepted in the

traceability community (Aizenbud-Reshef *et al.*, 2005; Olsen & Oldevik, 2007; Paige *et al.*, 2008; Walderhaug *et al.*, 2008). Defining link semantics in an enforceable way facilitates and supports richer analysis such as coverage analysis or orphan analysis (Walderhaug *et al.*, 2008). By link semantics, it is not necessarily meant formal semantics. What is meant is typed trace links accompanied by additional correctness constraints expressed in an appropriate language. This is the bare minimum for supporting automatic trace link manipulation by tools. By observing the various approaches under study only four out of the forty-five traceability approaches (8%) support links with rich semantics. Other approaches acknowledge the need for typed links (e.g. (Ramesh & Jarke, 2001)) without dealing with constraints, while others do not deal with the semantics of links at all. This is the case for example with the IR approaches such as (Hayes *et al.*, 2003) or (Antoniol *et al.*, 2002), which support the discovery of generic links between artefacts without dealing with the type of the discovered relationship.

One observation that can be made by studying the data matrix in Table B.2 is that there seems to be a negative relationship between an approach’s automation of the identification activity and its support for rich link semantics. Only two out of the forty-five different traceability approaches (4%) support both automatic traceability identification and rich semantics of trace links (Grammel & Voigt, 2009; Schwarz *et al.*, 2009). To validate this observation we have calculated the correlation between the two attributes using the *Cramèr’s v* correlation coefficient (Cramér, 1946), which is designed to measure the strength of association between categorical (nominal) data. The value of the correlation coefficient is 0.809. This indicates a strong correlation between the two variables. Since we are dealing with qualitative data, the notion of negative or positive correlation does not exist and hence can not be captured formally by the correlation coefficient. However, since our data matrix is small we can observe that in most cases when the *Automatic Traceability Identification* attribute is “present”, i.e. has a value of 1, the *Rich Link Semantics* attribute is “absent”. The only exceptions are the two approaches mentioned above. Table 3.5 summarizes the above discussion.

Table 3.5: Link Semantics in Relation to the Automation of the Identification Activity

	Manual Identification	Semi-automatic	Automatic	Total
Rich Link Semantics	4	0	2	6
No Rich Link Semantics	8	19	12	39
Total	12	19	14	46

Observation # 4 - Artefact Support: During the development and deployment of software systems, different heterogeneous artefacts are used. From informal models of the environment in which the system will operate, to source code and its documentation. These artefacts can have different formats. For example they can have graph-like

or tree-like structure or they can be unstructured text. Moreover, these artefacts are often represented in various heterogeneous notations. A generic traceability approach should be able to deal with different types of artefacts and notations. This can be achieved by having the appropriate extension mechanisms to accommodate the extra notations or artefacts. From the traceability approaches under study, eighteen of them (40%) are artefact specific. That is, the solution they propose applies only to particular combinations of artefacts. For, example, LeanArt (Grechanik *et al.*, 2007) supports traceability between UML use cases and source code. Although some of these approaches are quite efficient in supporting specific scenarios of traceability, the solutions they propose can not be easily generalised. This is due to the fact that the technologies they are using are artefact specific or that they are using artefact-specific characteristics to enable the capture of traceability.

Observation # 5 - Traceability Representation: In section 2.4.2, two main approaches have been identified for specifying the semantics of traceability information. Generic traceability metamodels attempt to capture the types of trace links, semantic expressiveness and other qualities of traces that should be record for the general case. Due to the diversity of situations in which traceability information needs to be captured, such metamodels can not achieve completeness. Hence, case-specific metamodels are developed in order to capture the structure of traceability information for particular scenarios. For the purposes of this analysis, we have broken down the case of case-specific metamodels into two subcategories. The first category includes approaches, which develop a case-specific metamodel for the scenario the approach is dealing with, i.e. for the notation the approach is considering or for the types of artefacts the approach deals with. This metamodel can apply only in the particular scenario and can not be extended (e.g. (Sherba *et al.*, 2003)). On the other hand, there are approaches which do not focus on specifying a case-specific traceability metamodel, but they focus on providing the mechanisms for specifying such metamodels with reduced effort (e.g. (Fabro *et al.*, 2005)). Out of the forty-five approaches considered in this analysis only eight (17%) provide a metamodel which can be used in different scenarios by extending it. Twelve approaches provide a metamodel only for the scenario they are dealing with and they do not suggest any extensions for different scenarios, while eleven approaches use a generic metamodel, which consists of the basic concepts of traceability such as *trace link* or *traceable artefact*. The rest of the approaches do not deal at all with specifying traceability information using a structured way such a metamodel or schema.

Observation # 6 - Traceability Maintenance: Traceability maintenance, i.e. the activity of maintaining the integrity of trace links while the referenced entities continue to change and evolve, is considered a crucial activity (Aizenbud-Reshef *et al.*, 2006a; Cleland-Huang *et al.*, 2002a). Despite of its importance however, there is an imbal-

ance in the literature between the approaches which cover the rest of the traceability activities and the approaches which cover traceability maintenance. Out of the forty-five approaches only six (13%) deal with this aspect of traceability. Moreover, these approaches do not consider any other traceability activities and some of them are very specialised. For example, (Mäder, 2008) develops a tool for maintaining trace links between UML class diagrams only.

In this section, a set of observations was made based on the data matrix, which was constructed during the phenetic analysis. To sum up the above findings, one can note that there are many different parameters which can affect the applicability of a traceability approach in different scenarios such as the different viewpoints it supports, the different traceability activities it covers or the different types of trace links it can distinguish. Based on the above observations, no traceability approach can be applied in every scenario. Most of the approaches focus on particular aspects of traceability and hence they are developed in isolation from the other approaches. This comes into contrast with the software development practices, where many intertwined technologies and development environments are used to develop heterogeneous types of interrelated artefacts. In such development scenarios, isolated approaches can not perform satisfactory since they are unable to cover the entire development effort. Moreover, they are not very easily integrated with other approaches, since they are not built with extensibility in mind.

3.3.2 Interpretation of the Dendrogram

Figure 3.1 is a tree structure, which represents the similarity between the different traceability approaches. This dendrogram can provide a system for conducting, documenting and coordinating a comparative study of traceability approaches. One aspect to be noted in this study is that this is a pioneering study, because a classification of traceability research approaches has never been studied using the phenetic approach before.

The first cluster created is the *Generative Approaches* cluster and consists of the following approaches:

- (Costa & da Silva, 2007)
- (Olsen & Oldevik, 2007)
- (Falleri *et al.*, 2006)
- (Amar *et al.*, 2008)
- (Rose *et al.*, 2008)
- (Grammel & Voigt, 2009)
- (Kolovos & Paige, 2010)
- (Schwarz *et al.*, 2009)
- (Jouault, 2005)
- (Vanhooff *et al.*, 2007a)

Since this cluster was created first, this indicates that the approaches belonging to

it demonstrate the highest similarity with each other. The focus of these approaches is on traceability as a by-product of model transformations. Such model transformations can be either model-to-model or model-to-text transformations. Generative traceability is achieved by either inserting traceability related code in the transformation code or by relying on the transformation engine to produce the traceability. The approaches, which rely on a transformation engine to generate traceability, do not support links with rich semantics while the approaches, which rely on additional traceability-specific code, tend to *pollute* the transformation with code not directly related to the transformation. *Generative Approaches* can generate trace links automatically. Moreover, all of the approaches in this cluster represent the traceability information in graphs or trees. Although all of the *Generative Approaches* can identify trace links automatically, none of them can support the automatic maintenance of the recovered links. Finally, none of the methods can support the Dependency viewpoint of traceability.

The next cluster consists of approaches which focus mainly on solving the problem of traceability representation by trying to define the structure of traceability information. Hence we name this cluster *Representation-intensive Approaches* cluster. The approaches in this cluster demonstrate the highest similarity with the approaches in the *Generative Approaches* cluster. This has to do mainly with the fact that nine out of the eleven approaches capture traceability information in graphs (i.e. models), while they define the structure of those graphs using traceability metamodels. More particularly, six approaches use case-specific metamodels, which can be extended to support different traceability scenarios, one approach uses a generic and non-extensible metamodel and three approaches define a generic metamodel. Moreover, seven out of the eleven approaches in this cluster support model-to-model traceability. On the other hand, the approaches belonging to this cluster do not provide any automation for the identification of traceability relationships and they support only the *Dependency* viewpoint of traceability. Finally, half of the approaches of this cluster attempt to automate the maintenance of the identified trace links. At this point though, it should be noted that although the approaches in this cluster focus on traceability representation, none of them supports a rigorous way to define the semantics of the traceability information, that is case-specific extensible metamodels accompanied by correctness constraints. The approaches belonging to this cluster are the following:

- (Maletic *et al.*, 2005)
- (Murta *et al.*, 2006)
- (Pinheiro, 2004)
- (Mäder, 2008)
- (Sharif & Maletic, 2007)
- (Spanoudakis & Zisman, 2005)
- (Sousa *et al.*, 2008)
- (Fabro *et al.*, 2005)
- (Bayerand & Widen, 2001)
- (Ramesh & Jarke, 2001)
- (Walderhaug *et al.*, 2006)

The third cluster is the *Hyperlink-Based Approaches* cluster. This cluster consists of the following approaches:

- (Maletic *et al.*, 2003)
- (Spanoudakis *et al.*, 2004)
- (Jirapanthong & Zisman, 2007)
- (Grechanik *et al.*, 2007)
- (Spanoudakis *et al.*, 2003)
- (Sherba *et al.*, 2003)

The main characteristic of this cluster is the fact that all of its approaches represent traceability using hyperlinks. As a consequence, the focus of those approaches is on text intensive artefacts since hyperlink technologies are traditionally associated with such artefacts. Out of the six approaches of this cluster two deal with text-to-text traceability, one deals with model-to-text traceability while three of them deal with both. Another consequence of using hyperlinks to represent trace links is the fact that the *Hyperlink-Based Approaches* do not support rich link semantics. One possible interpretation to this might be the fact that the medium used to represent trace links, i.e. hyperlinks, is limiting the capture of rich semantics. This is because the hyperlink system used does not support typed or semantic links (Frei & Stieger, 1995). Furthermore, the approaches belonging to this cluster deal with the automation of the identification of traceability relationships. Five of them do this by using indirect and direct rules while one approach uses run-time monitoring and program analysis. Finally, all of the *Hyperlink-Based Approaches* support only the Dependency Viewpoint.

The final cluster generated by the phenetic analysis is the largest one. This cluster consists of the following approaches:

- (Ying *et al.*, 2004)
- (Zimmermann *et al.*, 2004)
- (von Knethen & Grund, 2003)
- (Zisman *et al.*, 2003)
- (Egyed & Grunbacher, 2002)
- (Pohl, 1996a)
- (Hayes *et al.*, 2006)
- (Hayes *et al.*, 2003)
- (Hayes *et al.*, 2004)
- (Zou *et al.*, 2006)
- (Cleland-Huang *et al.*, 2005)
- (Lin *et al.*, 2006)
- (Antoniol *et al.*, 2002)
- (Natt och Dag *et al.*, 2005)
- (Marcus & Maletic, 2003)
- (Lucia *et al.*, 2008)
- (Pierce, 1978)

All of the approaches focus on text intensive artefacts hence we call this cluster *Text-Intensive Approaches* cluster. The approaches in the previous cluster focus as well on text-intensive artefacts. However, there are two main differences. First, while the approaches in the previous cluster use mainly rule-based techniques to automate the

identification of the trace links, the approaches belonging to this cluster use mainly IR techniques. Second, the methods of the previous cluster store traceability information in the artefacts it refers to (intra-artefact storage), while the methods of this cluster store traceability information in dedicated data structures. Thirteen approaches use matrices to store traceability information, two use graphs, while for two of the approaches is not mentioned in their documentation how traceability information is stored. Finally all of the approaches of this cluster deal only with the Dependency viewpoint.

To conclude, each of the traceability approaches under study can be considered as a collection of different attributes. Figure 3.1 is a tree structure, which represents the similarity between those approaches based on their attributes. This dendrogram can provide a system for conducting and documenting a comparative study of the various traceability approaches. Moreover the generated dendrogram can be used to understand more about the factors that are critical to a traceability approach. For example, it is clear that the majority of the traceability approaches focus mainly on pre-existing artefacts (Dependency Viewpoint), while the *Generative Approaches* ignore completely this aspect of traceability. This is due to the fact that the emphasis on the majority of those approaches is on model transformations, hence the focus is on the Generative Viewpoint. Finally, the generated dendrogram together with the data matrix can be used to highlight the research and knowledge gaps within the field of traceability. For example, one such gap can be the fact that the *Generative Approaches* focus only on the Generative Viewpoint of traceability, while it is argued in the literature that both the Dependency and the Generative viewpoints should be covered. Based on the analysis so far, in the next section we will state the research hypothesis for this work.

3.4 Research Hypothesis

As stated in chapter 1, the aim of this work is to provide traceability support in MDE. In this section, we will summarise the basic principles of MDE and we will relate them to the observations and analysis of the previous sections. This will allow us to set up the context of the research hypothesis as well as the research hypothesis of this work.

To summarize the discussion which has taken place in chapter 1 and section 2.7, we can state that MDE relies on two simple facts (Jouault *et al.*, 2009). First, any kind of system can be represented by models. Second, any model must conform to a metamodel. Within this context, models can be and are intended to be automatically processed by a set of operators. These two simple facts are illustrated in Figure 3.2. Based on those two facts, the main principles of MDE can be derived. These are the following:

- Models are promoted to first-class citizens and they are used extensively to raise the level of abstraction at which developers create and evolve software. Therefore

models are used to bridge the gap between the problem domain and the software implementation domain.

- MDE is based on system decomposition, that is, the division of a system into elements in order to improve comprehension of the system and the way in which it meets the needs of the user. Because of the limited capability of humans to understand complexity, a “divide and conquer” system decomposition approach is appropriate. Effective application of system decomposition requires the means of modeling the system from a variety of viewpoints and at increasing levels of specificity. Very often models for different viewpoints are expressed in different notations. This is desirable because it is much easier to define and share that view if it is expressed in a notation that is suited for that purpose (Egyed *et al.*, 2005).
- Every model has to conform to a metamodel. Metamodels are used as filters to define different viewpoints. Used as a typing system, they provide precise semantics to artefacts and relations between these artefacts. This homogeneity of definition of metamodels and models enable engineers to apply operations on them in a generic way.
- Since models are considered as first-class citizens, they “drive” the development process. This is achieved by using models to derive other models by performing different automatic model management operations such as transformation, composition and difference. The models, which are the product of such operations are called *derived models*. However other models have to come from somewhere in the first place. These models are usually created manually by engineers and they are called initial models.

These principles have an effect on the requirements of any traceability approach, which targets MDE processes. The requirements for such a traceability solution are the following:

1. **Requirement 1:** Since models are the primary artefacts used in MDE, the traceability approach should be able to support traceability between models. This is not enough however, since when applying MDE, development of initial models starts often from other kinds of artefacts such as informal, natural language descriptions of requirements and spreadsheets, and ends up to non-model artefacts such as source code and system documentation. Hence, any traceability solution in MDE needs to consider these artefacts as well, in terms of how models can be traced to other (non-model) artefacts and how (non-model) artefacts can be traced to models (Paige *et al.*, 2008).
2. **Requirement 2:** Given that it is common that different notations are used to express different viewpoints (Egyed *et al.*, 2005), a traceability approach in MDE

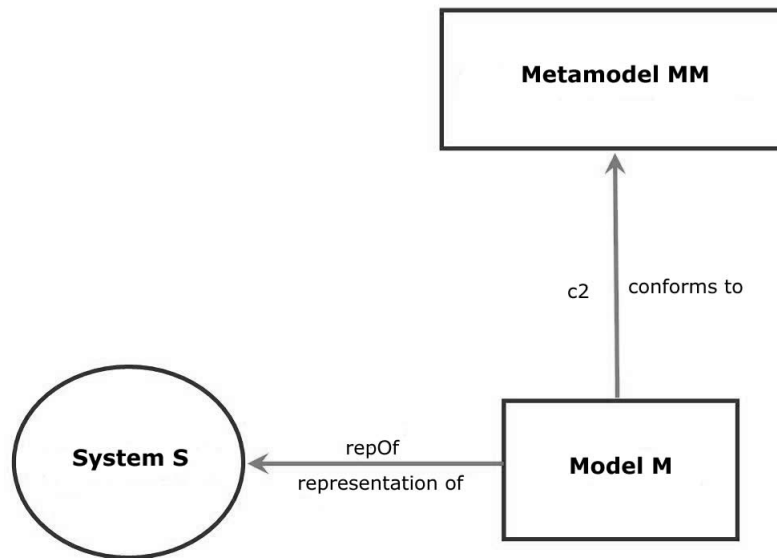


Figure 3.2: Basic relations of representation and conformance in MDE (adapted from (Jouault *et al.*, 2009))

should not be notation specific but it should be either generic or extensible in order to facilitate the different notations.

3. **Requirement 3:** Since models conform to different metamodels and have different semantics, traceability information between different models can possibly have different semantics. Moreover, depending on the use of traceability different semantics might apply to traceability information between the same models. Hence, a traceability approach should be able to define the semantics of traceability information for the various traceability scenarios. As a result, traceability information can be automatically manipulated and analysed by a machine.
4. **Requirement 4:** In view of the fact that, models can be generated automatically by applying model operations on other models (derived models) or manually by engineers (initial models), a traceability approach should be able to support both the Generative and Dependency viewpoints of traceability.

Apart from these four requirements, we have identified two more based on the analysis conducted in the previous sections. These requirements are not directly related with the application of traceability in MDE, but they could benefit any traceability approach. These requirements are the following:

5. **Requirement 5:** A traceability approach should cover the four basic traceability activities described in Chapter 2. If an approach focuses on some of the traceabil-

ity activities ignoring the rest, then there is the danger that this approach uses not easily combinable technologies and techniques. As a result, combining different approaches to cover all aspects of traceability can be a challenge. Therefore, a traceability approach should either support all of the basic traceability activities or provide the appropriate extensibility mechanisms for the activities it does not support.

6. **Requirement 6:** A traceability approach should automate as much as possible the different traceability activities. By automating traceability activities, the cost and complexity of traceability is reduced (Egyed *et al.*, 2005). At this point however it should be noted that not all of the traceability activities can be automated fully. Traceability representation for example is an activity which relies on human intuition and knowledge of a domain, hence it can not be automated. Moreover, traceability activities can not be fully automated for every possible scenario. This is due to the fact that establishing and maintaining traceability often relies on informal information. Due to this fact, there is always a trade off between semantics and automation. Fully automatic identification and maintenance of traceability can not be achieved without ignoring the informal information, which is not understandable by a machine.

The above requirements can be used to derive the thesis hypothesis:

This thesis demonstrates that a domain specific model-based traceability approach can support and automate the process of rigorously managing the different types of heterogeneous traceability relationships between both derived and initial models in an MDE process.

The main characteristics highlighted in the above statement are the following:

1. **Model-based Approach:** In the spirit of MDE the proposed approach shall be applicable to model artefacts.
2. **Manage Traceability:** The proposed approach will provide support for all four traceability activities, i.e. representation, identification, maintenance and usage of traceability.
3. **Heterogeneous Traceability Relationships:** Using the proposed approach, an engineer will be able to manage traceability relationships between models expressed in heterogeneous notations.
4. **Rigorous:** The proposed approach will support the capture of case- specific or scenario-specific semantics.

5. **Automation:** The proposed approach will provide semi-automatic identification and maintenance of traceability information
6. **Derived and Initial Models:** The proposed approach will be suitable for traceability between models, which are generated automatically by applying model operations on other models (derived models) as well as between models, which are created manually by engineers (initial models).

Based on the previous discussion, the objectives of this research are the following:

- To propose an approach with which rigorous, well-defined, case-specific traceability models can be developed.
- To support and automate the activity of identifying traceability links between models.
- To support and automate the activity of maintaining traceability information between models.
- To propose novel usage scenarios of traceability information in MDE.

The objectives of this thesis - listed above - should fulfil the previously identified requirements for traceability support for MDE processes.

3.5 Research Scope

The purpose of this section is to establish the scope and boundaries of this work. As described in the previous sections traceability support in MDE should be able to accommodate both models and textual artefacts. Due to the fact that dealing with traceability for models and for textual artefacts is quite different, as well as given the time and resource constraints of this work, a decision has been made to limit the scope of this work to address only the technical space of models. This decision was made on the grounds that models are the main artefacts in MDE and it is them that guide the development process.

3.6 Research Methodology

The main purpose of this section is to provide a description of the methodology, which was used in order to evaluate the validity of the hypothesis. To this end, a typical iterative software engineering process was used. The process consisted of multiple iterations

of an analysis phase, a design phase, an implementation phase and a testing phase. A graphical overview of this process is illustrated in Figure 3.3. In the following, the four phases of the process are briefly described.

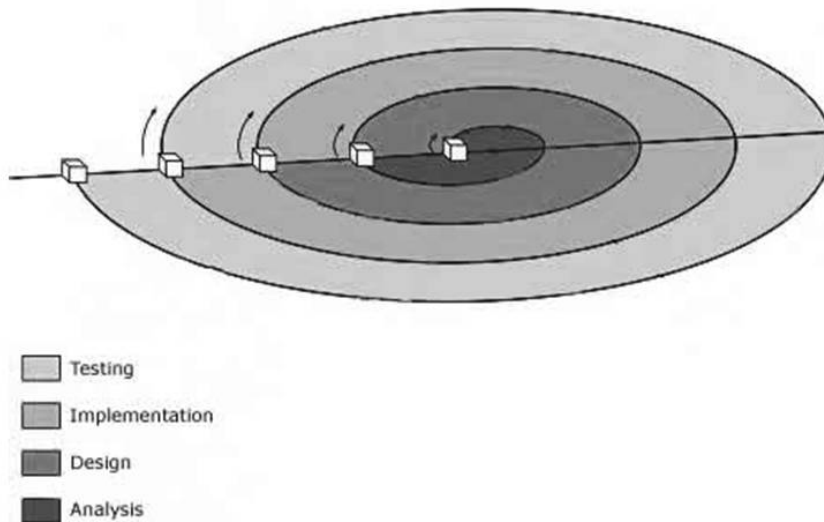


Figure 3.3: Overview of the research methodology

3.6.1 Analysis

In the analysis phase, the most influential and the most well-documented approaches to traceability have been identified. These approaches have then been reviewed in terms of the technical aspects, the assumptions made about traceability as well as the advantages and the shortcomings. Through this analysis a number of research challenges has been identified that have motivated the hypothesis and objectives of this thesis.

3.6.2 Design and Implementation

Based on the findings of the analysis phase, a conceptual solution has been designed in order to investigate the hypothesis. Through multiple iterations, this designed has been

refined to a technical design, which in turn has guided the development of the reference implementation.

3.6.3 Testing

Throughout the previous phases, case studies have been identified and used in order to assess the quality of the proposed approach as well as the correctness of the reference implementation. Further iterations of the process have been guided by findings in the testing phases.

3.7 Chapter Summary

In the first part of this chapter, a detailed analysis of the literature review took place. This analysis took place using a method called Numerical Taxonomy. This method is utilised by biologists in order to classify living organisms. Based on this analysis the research challenges of this work were identified. Moreover, the research hypothesis of this thesis was established and the objectives have been identified. The following chapters discuss the establishment of a domain-specific and model-based approach to traceability in MDE. This approach is used as means for exploring the proposed hypothesis.

Chapter 4

A Metamodelling Approach to Traceability

In the previous sections, traceability approaches found in the literature have been presented. Based on this extended literature survey, these approaches have been classified using the method of numerical taxonomy. This analysis has led to the identification of a set of requirements for a traceability approach in the context of MDE. In this section, a novel approach to model-to-model traceability support is presented. This approach focuses on how to define and implement semantically rich trace links that are amendable to automatic manipulation.

At the crux of the proposed approach lies the Traceability Metamodelling Language (TML), which is a domain-specific metamodelling language (Zschaler *et al.*, 2009) for traceability. The purpose of TML is to enable the construction of case-specific and semantically rich traceability metamodels as well as the generation of their accompanying correctness constraints. This rigorously defined traceability information can be then used to automatically identify trace links between models and to automatically maintain the established links. Finally, the recovered trace links can be used in a variety of different ways depending on the traceability scenario. In other words, the approach proposed in this work covers all four activities of the traceability lifecycle, namely representation, identification, maintenance and usage.

A fundamental assumption of the proposed approach is that in the general case one needs to establish trace links between elements belonging to a number of models that potentially conform to diverse metamodels. Additionally, several types of trace links linking different types of model elements may need to be captured, depending on the traceability scenario. Finally, correctness constraints that extend beyond simple type conformance should be captured. These constraints are usually domain and/or case-specific. As a consequence, the traceability engineer might need to specify multiple

traceability models which conform to different traceability metamodels. The aim of TML is to enable the construction and maintenance of traceability metamodels and the accompanying constraints with reduced effort by encoding constructs and relationships that are often encountered when constructing traceability metamodels into first-class metamodeling artefacts. Moreover, TML can be used for generating the appropriate supporting infrastructure of the generated metamodels. A conceptual diagram of the holistic approach to traceability proposed by TML is illustrated in figure 4.1.

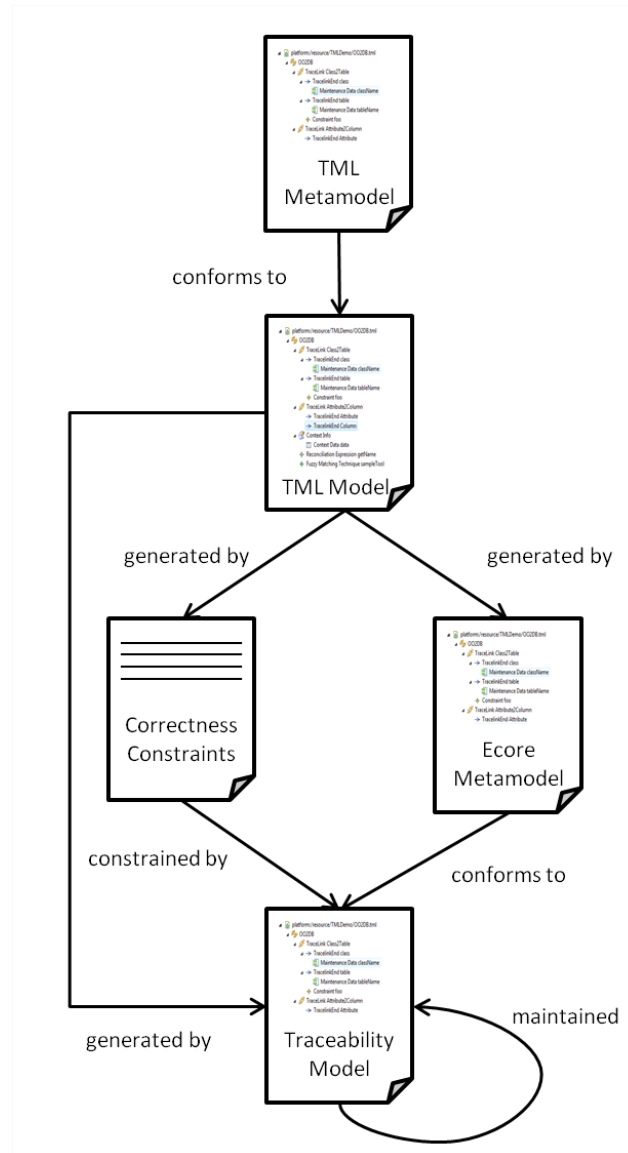


Figure 4.1: Conceptual diagram of the TML approach

For a given traceability scenario, a developer needs to capture valid traceability information for the metamodels of interest in a TML model. The information contained in the TML model are then used to generate other artefacts or to support the various traceability activities. An Ecore metamodel and its accompanying correctness constraints can be generated by the TML model. These artefacts are directly related to the traceability representation activity and define the valid traceability relationships for a given scenario. Moreover, a TML model can be used to recover trace links between two models, which conform to the metamodels the TML model refers to, in order to support the traceability identification activity. Moreover, the traceability model, which consists of the various recovered links, can be evolved and maintained reflectively by using the case-specific maintenance information captured in it.

This chapter consists of four main sections. In section 4.1, the abstract syntax and semantics of TML are presented. Section 4.2 discusses how TML models can be used to recover trace links between two models. Section 4.3 provides a discussion on how traceability models can be maintained in a valid state, while the models to which they refer change. Finally, in section 4.4, two novel traceability usage scenarios in the context of MDE are presented.

4.1 Representation of traceability with TML

As discussed in the beginning of this chapter, any traceability scenario in the context of TML should start by defining the traceability information for the given scenario. That is, the traceability engineers should define the various entity types to be traced and the relationship types between them. Moreover, they should decide on the data structures with which they will represent the captured traceability information as well as how such information would be visualised and stored.

It is widely accepted in the literature that the meaning of trace links should be precisely defined (e.g. (Aizenbud-Reshef *et al.*, 2006a; Paige *et al.*, 2008; Walderhaug *et al.*, 2008)). A precise semantics for traceability will allow developers to capture more accurately the intended meaning of a specific traceability relationship, and will enable richer, tool-supported analysis and reasoning about traceability. In the TML approach, semantically rich trace links should possess two characteristics:

1. They should be strongly typed conforming to a case-specific traceability metamodel.
2. The case-specific metamodel should be accompanied by a set of case-specific correctness constraints, which express validity requirements that can not be captured by the metamodel itself.

These characteristics are sufficient for supporting rich traceability in the sense that we discussed in the previous sections of this work. In the following, we will explain briefly why these characteristics are important and sufficient for supporting trace links amendable to manipulation by traceability tools.

4.1.1 Strongly Typed Trace-links Conforming to a Case-Specific Metamodel

To support *rigorous* traceability, trace links should be strongly typed. In this way, the various semantic heterogeneities among the different trace-links can be captured, hence richer and more precise analysis and reasoning (human or computerized) can be facilitated. Additionally, creation of illegitimate links can be prevented. For example, consider the case where we need to define a trace metamodel which allows the establishment of trace-links between instances of *A* from metamodel *MMa* and instances of *B* from metamodel *MMb*, but no links between two instances of *A* or two instances of *B*. By specifying the strongly typed link *A-to-B*, the user avoids the generation of the invalid link between two instances of *A* or two instances of *B*. These link types are specific to creating mappings between metamodels *MMa* and *MMb* and have a meaning only in this context.

In the spirit of MDE, the models which contain the traceability information should conform to a metamodel. The argument for this is uniformity and standardisation: if traceability information is also a model, then standard MDE tooling can be used to process (e.g., transform, validate, analyse visualise) this information. To accomplish this, there are two alternative approaches: use of a general purpose traceability metamodel or use of multiple case-specific traceability metamodels. In the case where a general purpose trace metamodel is used, a trace link can connect any number of elements, of any type and in any model.

Instead of defining only generic trace link types in a general-purpose traceability metamodel, one could capture all possible existing link types in a comprehensive metamodel. However, attempting to this is not feasible since the space of trace links is vast (Paige *et al.*, 2008). Most of the time the decision of which link types to use depends on the particular usage scenario as well as on the domain. Therefore, it is argued in the literature (Jouault, 2005; Kolovos *et al.*, 2006c), that trace-link types should be defined in a case-specific/scenario-specific trace metamodel, in order to enable richer, case-specific analysis. To specify the trace-link types, the trace metamodel needs to explicitly refer to types of elements defined in other metamodels, namely the metamodels a particular trace link refers to. For the definition of such a traceability metamodel, a modelling technology that does not consider each metamodel as a closed space i.e. a technology which supports inter-metamodel references must be employed. An exemplar modelling technology which has this functionality is the Eclipse Modelling Framework (EMF)

(Eclipse Foundation, 2010a).

An example of case-specific trace information is the *link rationale* associated with particular types of trace-links. A description of a trace-link's *raison d'être*, as well as the assumptions made behind its creation, can provide additional information about its validity. The rationale behind a trace-link should capture why a link exists, the assumptions under which the link is valid, and the various alternatives and argumentation behind the choice of one of those alternatives. Additionally, link rationale can support accountability of trace-links. While this concept could be represented in a general-purpose trace metamodel (i.e., as a metaclass), such an approach would not let the developers easily encode case-specific information that supports rigorous analysis – they would have to encode case-specific information using general concepts such as strings, or other primitive datatypes, thus making analysis more difficult to carry out.

Despite the fact that case-specific trace metamodels require additional effort for their construction and also extra effort to support the direct communication of tools with heterogeneous trace metamodels, we argue that the definition of such metamodels can provide flexibility and specificity for supporting analysis, in a way that is beneficial to a system development project. The main benefit of this approach is its ability to capture domain information and semantics, suitable for different scenarios.

4.1.2 Correctness Constraints

Apart from the aforementioned type-safety, there are often additional constraints that need to be specified and which the trace metamodel can not capture by itself. For example, in the context of the previous example, an additional constraint may dictate that for each instance of A from MMa there exists one and only one A -to- B link, that links it with an instance of B from MMb . Such a constraint can be specified using a constraint language which supports the expression of constraints spanning over elements belonging to various models of potentially heterogeneous metamodels. The Object Constraint Language (Object Management Group, 2003) currently lacks such capabilities as it does not provide constructs for expressing inter-model constraints. Exemplar constraint languages that support establishing intra-model constraints include the Epsilon Validation Language (EVL) (Kolovos *et al.*, 2007) and the XLinkit toolkit (Christian Nentwich, Licia Capra, Wolfgang Emmerich and Anthony Finkelstein, 2002).

The combination of a strongly typed case-specific trace metamodel with inter-model constraints that can be checked automatically prevents users and tools from establishing incorrect trace-links. Furthermore, such an approach enables trace-links that can be automatically validated and analyzed to discover potential omissions and inconsistencies. Such inconsistencies can arise either during the establishment of the trace-links or later on in the lifecycle of the models among which traceability links have been established.

Through experimentation with defining a number of case-specific traceability meta-

models we have identified a set of recurring patterns both in terms of the structure of the metamodelling and in terms of the additional constraints that guarantee the semantic integrity and completeness of the trace models. A description of these recurring patterns can be found in (Drivalos *et al.*, 2008). The existence of recurring patterns hints that a higher-level of abstraction is potentially beneficial for defining trace metamodelling. By this rationale, we have developed the Traceability Metamodelling Language (TML) (Drivalos *et al.*, 2008), a domain specific language which promotes the identified patterns into first-class artefacts and which is dedicated to the construction of trace metamodelling with the aforementioned characteristics.

4.1.3 Recurring Patterns in Case-Specific Traceability Metamodelling

In section 4.1.1, we have argued for the need for traceability models that conform to case-specific metamodelling. Through experimentation with defining a number of case-specific traceability metamodelling we have identified a set of recurring practices and patterns both in terms of the structure of the metamodelling and of the additional constraints that guarantee the semantic integrity and completeness of the traceability models. In this section, we outline the identified patterns and demonstrate them through a simple – but representative – example. In our example, which is based on the example originally demonstrated in (Drivalos *et al.*, 2008), we have hand-crafted a traceability metamodelling (*ComponentClassTraceMetamodelling*) for capturing traceability links between models that conform to two minimal *ClassMetamodelling* and *ComponentMetamodelling* metamodelling.

The three metamodelling appear in figure 4.2. The aim of this example is to demonstrate how traceability information can be captured between a component model and a class model. Such a scenario can arise in a Component-Based Development Environment, where class diagrams are used to refine the architecture specified by component diagrams into a concrete design. In this example, we use two simple metamodelling, the *ClassMetamodelling* and the *ComponentMetamodelling*, which are illustrated at the top and bottom of figure 4.2 respectively. Our aim is to capture traceability links between models which conform to those two metamodelling, i.e. the *ComponentModel* and the *ClassModel*. The trace model which will hold the trace information is the *ComponentClassTraceModel* and conforms to the *ComponentClassTraceMetaModel*. More precisely, we want to trace instances of Package from the *ClassMetamodelling* and instances of Component from the *ComponentMetamodelling*. Additionally, we want to capture links between instances of Method from the *ClassMetamodelling* and instances of Service from the *ComponentMetamodelling*.

Through the process of defining traceability metamodelling similar to the one presented in figure 4.2 we have identified that each traceability metamodelling contains a root class (typically named *TraceModel*) that acts as the root of a traceability model. Moreover, it contains a number of traceability link types (*ComponentPackageTraceLink* and

4.1 Representation of traceability with TML

ServiceMethodTraceLink in our example), all of which extend an abstract *TraceLink* abstract class for structuring purposes. Each traceability link contains a number of references of different types and cardinalities. For example, the *ComponentPackageTraceLink* metaclass contains the one-to-one *component* reference and the one-to-one *package* reference so that it is able to trace one component to one package only. On the other hand, the *ServiceMethodTraceLink* metaclass contains the one-to-one *service* reference and the one-to-many *methods* reference so that it can trace one *service* to multiple *methods*.

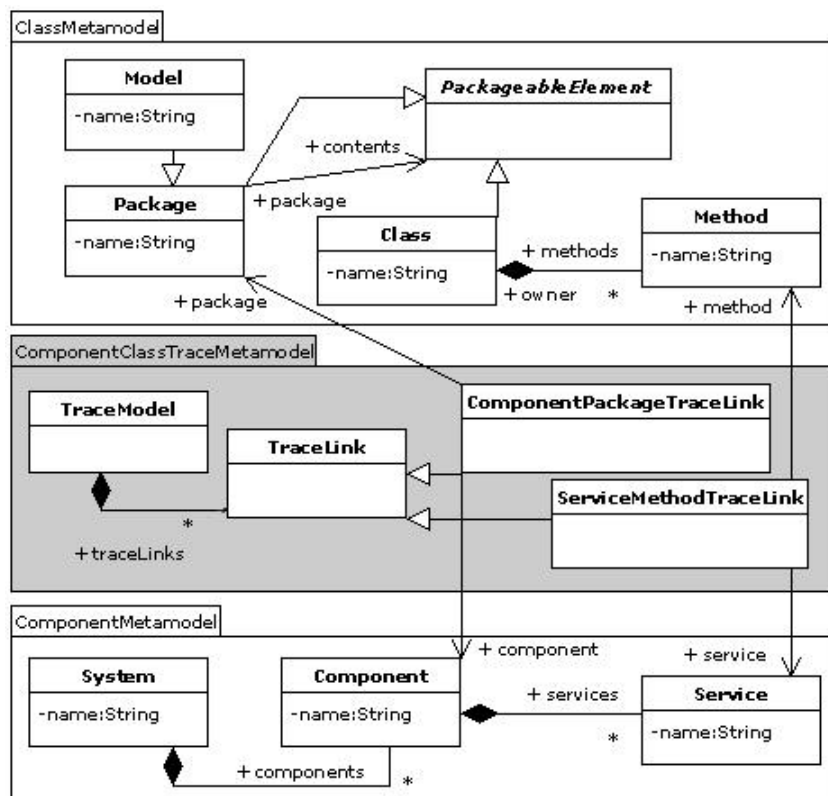


Figure 4.2: The ComponentClassTraceMetamodel Traceability Metamodel

Apart from the references that represent link ends (e.g. *packages*, *services*), each traceability link also typically stores some additional primitive information. This information either applies to all traceability links (e.g. the *description* and *creator* attributes in meta-class *TraceLink*) or to some particular types of traceability links only (e.g. the *alternatives* attribute in the *ServiceMethodTraceLink* meta-class which shows if the service depends on all of the methods (logical and) or only on one of them (logical or)).

Moreover, establishing a traceability metamodel extends beyond constructing the abstract syntax and involves also the specification of validity constraints. Examples of

such constraints are the following:

ForAll: It is often required that at least one traceability link of a particular kind exist for all instances of a type. In our example, we require that at least one service-method traceability link exists for each service in the *Component* model.

Unique: Another common constraint is when it is required that at most one traceability link of a particular kind exists for each instance of a given type. In our example it is required that each *component* can link to at most one *package* to avoid fragmentation.

Optional: Additional information captured in the form of primitive attributes (e.g. *description*) is often neglected by end-users, thus resulting in poorly documented traceability models. For this purpose, it is typical that additional constraints are specified that ensure that the values of the attributes in the trace models are not empty. Apart from these reoccurring patterns, there are other case-specific constraints that cannot be generalized. For instance, in our example such a constraint is that *the methods a service traces to belong to classes contained in a package which is traced to the component that provides the service*.

The existence of reoccurring patterns hints that a higher-level of abstraction is potentially beneficial for defining traceability metamodelling languages. With this rationale, in the next section, we attempt to promote the identified patterns into first-class artefacts in a metamodelling language dedicated to the construction of traceability metamodelling languages.

4.1.4 The Traceability Metamodelling Language

Once recurring patterns, such as those discussed above, have been identified, the next step is to derive more abstract constructs from them. These constructs are encoded in the Traceability Metamodelling Language (TML). TML enables the construction and maintenance of traceability metamodelling languages and their accompanying constraints with reduced effort by encoding constructs and relationships that are often encountered when constructing traceability metamodelling languages into first-class metamodelling artefacts.

In three-level metamodelling architectures such as MOF and EMF, there is only one metamodelling language and that is MOF (Object Management Group, 2011a) and Ecore (Eclipse Foundation, 2010a) respectively. As such, defining a metamodelling language in the context of such an architecture appears to be a contradiction. Therefore at this point we should clarify that strictly speaking, TML is not a metamodelling language but a modelling language, since its metamodel is expressed using Ecore. However, TML models (instances of the TML metamodel) are eventually transformed into Ecore metamodels, which conceptually renders TML a metamodelling language. This is further clarified in the sequel.

4.1 Representation of traceability with TML

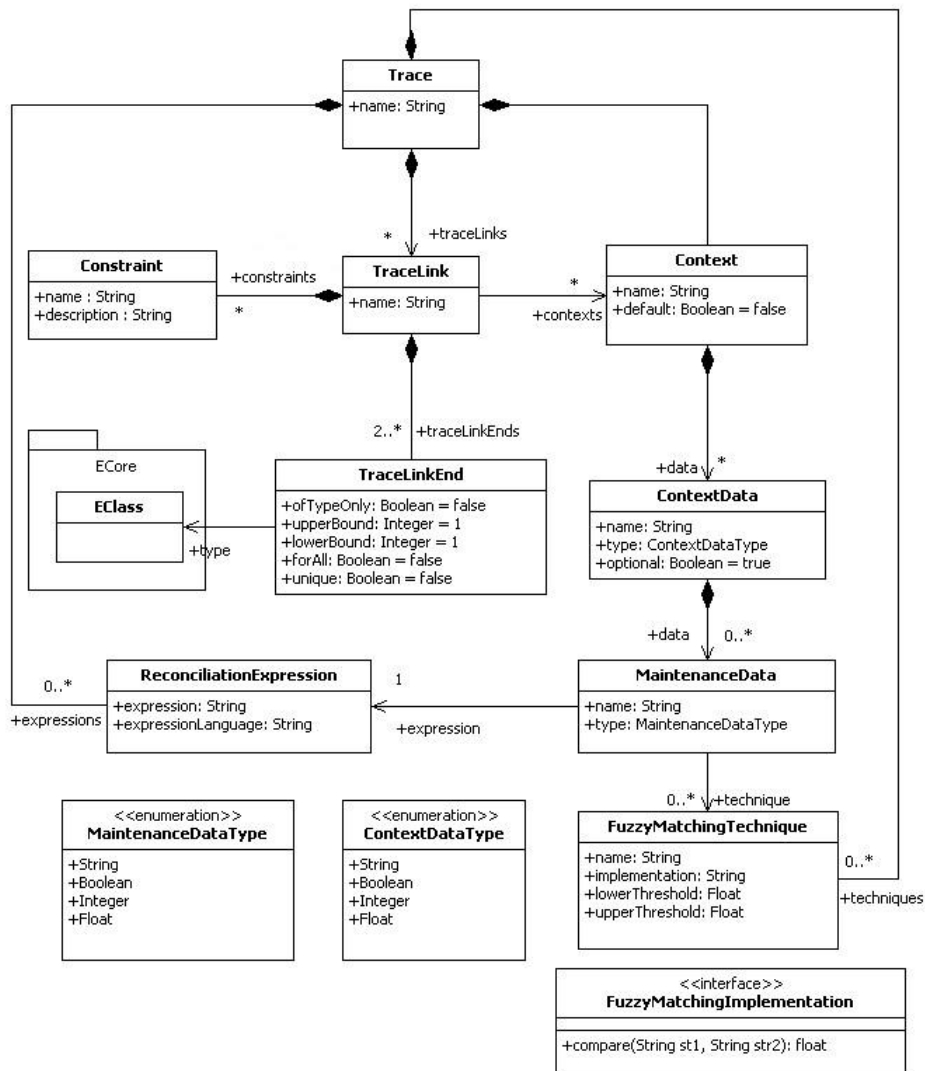


Figure 4.3: Abstract Syntax of TML

4.1.4.1 Abstract Syntax

As discussed above, the abstract syntax of TML has been defined using Ecore and is presented in Figure 4.3. In this section, the concepts of the TML metamodel are discussed in detail.

Trace: Acts as the root of a TML model. A Trace defines a name and can contain a number of TraceLinks and Contexts. Moreover, it can contain FuzzyMatchingTech-

niques and ReconciliationExpressions.

TraceLink: Represents a traceability link between a number of elements and it has a TraceLinkType. It defines a name and contains a number of TraceLinkEnds. It also associates to a number of contexts through which it can capture custom information. Finally, it can be associated to a number of Constraints, which ensure the semantic correctness of a TraceLink.

TraceLinkEnd: Represents an end of a traceability link. It defines a name and the type of elements it can link to. Through the *ofTypeOnly* attribute it also defines if it can link to elements of the particular type only or if it can also link to instances of all the subtypes of the type. The *unique* attribute defines if more than one traceability link linking to the same model element of this type are legitimate. The *forAll* attribute specifies if it is mandatory for all elements of that type to exist a traceability link of this kind. The *lowerBound* and *upperBound* attributes specify how many elements of the specified type can be linked to the same end.

Constraint: Represents case-specific correctness constraints that can not be automatically generated by the traceability metamodel. More information about these constraints will be provided in Section 4.1.4.2.

Context: The role of the context metaclass is to enable traceability metamodel designers to attach custom information to traceability links. Each context defines a number of *ContextData* and can be specified as *default*, which means that it is implicitly associated to all *TraceLinks* (as opposed to the explicit *data* reference of class *TraceLink*).

ContextData: Captures additional information of the primitive type (currently the String, Boolean, Float and Integer types are supported) about a *TraceLink*. It defines a name, a type and whether it is optional or not.

MaintenanceData: This meta-class represents data that must be captured in order to facilitate the automatic maintenance of a traceability model. This data can be retrieved by utilising dedicated ReconciliationExpressions.

ReconciliationExpression: Represents a string expression in a model query language such as the Epsilon Object Language (EOL). This expression can be used to query the models a traceability model refers to and retrieve particular data, which are important for the maintenance of the traceability model. More information about the ReconciliationExpression and the traceability maintenance mechanism built in TML in general will be provided in Section 4.3.

FuzzyMatchingTechnique: This meta-class is also associated with the maintenance of the traceability models and more particularly with the use of fuzzy matching techniques. A thorough discussion about how fuzzy matching can be used to recover broken trace links is provided in Section 4.3.

MaintenanceDataType: Provides the valid data types of MaintenanceData.

ContextDataType: Provides the valid data types of ContextData.

FuzzyMatchingImplementation: This interface can be implemented by a Java class in order to define different fuzzy matching algorithms for traceability maintenance. This is an extension mechanism of TML and it will be explained in section 4.3.

4.1.4.2 Semantics

The semantics of TML are specified in a translational manner (Kleppe, 2007) using two formal and executable transformations. These transformations transform a TML model into an Ecore metamodel and a set of constraints expressed in the Epsilon Validation Language (EVL) (Kolovos *et al.*, 2007).

Transforming to an Ecore metamodel As discussed above, a TML model lives in the M1 level of the three-tiered metamodeling architecture, and thus it cannot be instantiated as-is. Therefore, in order to instantiate it we need to transform it into an M2 level model (proper metamodel). The listings following demonstrate the transformation from TML to Ecore using the Epsilon Transformation Language (ETL) (Kolovos *et al.*, 2008), although in principle any model-to-model transformation language (e.g. ATL (Jouault *et al.*, 2006), QVT (Object Management Group, 2002), Kermeta (Chauvel & Fleurey, 2010)) could have been used for this purpose. In this section we go through the transformation and discuss its functionality. The complete transformation can be found in Appendix D.

Listing 4.1: The TML2Ecore Transformation expressed in ETL - *Pre* Block

```
1 pre {
2   var traceModel := new ECore!EClass;
3   traceModel.name := 'TraceModel';
4   var traceContext := new ECore!EClass;
5   traceContext.name := 'TraceContext';
6   traceContext.`abstract` := true;
7   var contextsReference :=new ECore!EReference;
```

4.1 Representation of traceability with TML

```
8 contextsReference.name := 'contexts';
9 contextsReference.eType := traceContext;
10 contextsReference.containment := true;
11 contextsReference.upperBound := -1;
12 traceModel.eStructuralFeatures.add(contextsReference);
13 var traceLink := new ECore!EClass;
14 traceLink.name := 'TraceLink';
15 traceLink.`abstract` := true;
16 var traceLinkEnd := new ECore!EClass;
17 traceLinkEnd.name := 'TraceLinkEnd';
18 traceLinkEnd.`abstract` := true;
19 var modelLinksReference := new ECore!EReference;
20 modelLinksReference.name := 'links';
21 modelLinksReference.eType := traceLink;
22 modelLinksReference.containment := true;
23 modelLinksReference.upperBound := -1;
24 traceModel.eStructuralFeatures.add(modelLinksReference);
25 var traceLinkEndStatus = new ECore!EClass;
26 traceLinkEndStatus.name = "TraceLinkEndStatus";
27 traceLinkEndStatus.`abstract` = true;
28 var traceLinkEndOKStatus = new ECore!EClass;
29 traceLinkEndOKStatus.name = "TraceLinkEndOKStatus";
30 traceLinkEndOKStatus.eSuperTypes.add(traceLinkEndStatus);
31 var traceLinkEndInvalidStatus = new ECore!EClass;
32 traceLinkEndInvalidStatus.name = "TraceLinkEndInvalidStatus";
33 traceLinkEndInvalidStatus.eSuperTypes.add(traceLinkEndStatus);
34 var traceLinkEndAmbiguousStatus = new ECore!EClass;
35 traceLinkEndAmbiguousStatus.name = "TraceLinkEndAmbiguousStatus";
36 traceLinkEndAmbiguousStatus.eSuperTypes.add(traceLinkEndStatus);
37 var traceLinkEndAmbiguousStatusCandidates = new ECore!EReference;
38 traceLinkEndAmbiguousStatusCandidates.name = `candidates`;
39 traceLinkEndAmbiguousStatusCandidates.upperBound = -1;
40 traceLinkEndAmbiguousStatusCandidates.lowerBound = 2;
41 traceLinkEndAmbiguousStatus.eStructuralFeatures.add(
    traceLinkEndAmbiguousStatusCandidates);
42 traceLinkEndAmbiguousStatusCandidates.eType = EcoreM2!EClass.all.
    selectOne(c|c.name = `EObject`);
43 var traceLinkEndStatusReference = new ECore!EReference;
44 traceLinkEndStatusReference.name = `status`;
45 traceLinkEndStatusReference.eType = traceLinkEndStatus;
46 traceLinkEndStatusReference.lowerBound = 1;
```


4.1 Representation of traceability with TML

```
47  traceLinkEndStatusReference.lowerBound = 1;
48  traceLinkEnd.eStructuralFeatures.add(traceLinkEndStatusReference);
49  traceLinkEndStatusReference.containment = true;
50 }
```

In Listing 4.1, the preamble block of the transformation between TML and Ecore is illustrated. In ETL, the *pre* block is a block of statements which are executed before the transformation rules are executed. In the above transformation, the purpose of the *pre* block is to initialize a traceability metamodel and create the main meta-classes. Lines 2 and 3 of the transformation create the container of a traceability metamodel, namely the *TraceModel* EClass.

In lines 4–12, the abstract class *TraceContext* is created and is added to the *TraceModel*. Moreover, the cardinality of the EReference associated with the *TraceContext* EClass is set to -1 meaning that in any traceability metamodel, there can be infinite contexts. All these contexts will extend the abstract *TraceContext* EClass. Lines 13–49 of the transformation generate the *TraceLink* abstract class, which is the root-type for links. Moreover, these lines specify that a *TraceModel* contains zero or more *links*. In addition to the *TraceLink* EClass, lines 13–49 create the *TraceLinkEnd* EClass as well as three EClasses, which are associated with the status of a link end, namely *TraceLinkEndOKStatus*, *TraceLinkEndInvalidStatus*, *TraceLinkEndAmbiguousStatus*. These EClasses are associated with the maintenance of the traceability model. If a link end is flagged to be ambiguous, it can have a set of 2 or more alternatives for this link end. This relationship is specified in lines 37–42.

Listing 4.2: Trace2EPackage Rule

```
1  rule Trace2EPackage
2  transform s : Trace!Trace
3  to t : ECore!EPackage {
4    t.name := s.name;
5    t.nsURI := s.name;
6    t.nsPrefix := s.name;
7    t.eClassifiers.add(traceModel);
8    t.eClassifiers.add(traceLink);
9    t.eClassifiers.add(traceContext);
10   t.eClassifiers.add(traceLinkEnd);
11   t.eClassifiers.add(traceLinkEndStatus);
12   t.eClassifiers.add(traceLinkEndOKStatus);
13   t.eClassifiers.add(traceLinkEndInvalidStatus);
14   t.eClassifiers.add(traceLinkEndAmbiguousStatus);
```

4.1 Representation of traceability with TML

```
15     t.eClassifiers.addAll(s.links.equivalent());
16     t.eClassifiers.addAll(s.contexts.equivalent());
17     for (c in s.links) {
18         t.eClassifiers.addAll(c.ends.equivalent());
19     }
20 }
```

Rule *Trace2EPackage* illustrated in Listing 4.2 creates a new *EPackage* from the TML Trace, sets its name and namespace URI and adds the EClasses generated from the *traceLinks* and *contexts* of the *Trace* to the list of *EClassifiers* of the *EPackage*. Furthermore, it adds the EClasses associated with the status of the link ends to the list of the *classifiers* of the *EPackage* and it adds the respective link ends to the appropriate *links* (lines 17 and 18).

Listing 4.3: *TraceLink2EClass* & *TraceLinkEnd2EClass* Rules

```
1 rule TraceLink2EClass
2   transform s : Trace!TraceLink
3   to t : ECore!EClass {
4     t.name := s.name + 'TraceLink';
5     t.eSuperTypes.add(traceLink);
6     for (c in s.contexts) {
7       var ref := new ECore!EReference;
8       ref.eType := c.equivalent();
9       ref.name := c.name.firstToLowerCase();
10      ref.upperBound := -1;
11      t.eStructuralFeatures.add(ref);
12    }
13    for (c in s.ends) {
14      var ref := new ECore!EReference;
15      ref.eType := c.equivalent();
16      ref.name := c.name;
17      ref.lowerBound := c.lowerBound;
18      ref.upperBound := c.upperBound;
19      ref.containment := true;
20      t.eStructuralFeatures.add(ref);
21    }
22  }
23
24 rule TraceLinkEnd2EClass
```

4.1 Representation of traceability with TML

```
25 transform s : Trace!TraceLinkEnd
26 to t : ECore!EClass {
27     t.name := s.eContainer().name + s.name.firstToUpperCase() + '
        LinkEnd';
28     t.eSuperTypes.add(traceLinkEnd);
29     t.eStructuralFeatures.addAll(s.maintenanceData.equivalent());
30     var ref := new ECore!EReference;
31     ref.eType := s.type;
32     ref.name := "target";
33     ref.lowerBound :=1;
34     ref.upperBound := 1;
35     t.eStructuralFeatures.add(ref);
36 }
```

In Listing 4.3, rule *TraceLink2EClass* creates a new *EClass* for each *TraceLink* and sets its name to be the same as the *TraceLink*. Moreover, it creates a new *EReference* for each link *end* of the *TraceLink* with respective cardinality (upper and lower bounds) (lines 13–20). Finally, rule *TraceLink2EClass* creates a new *EReference* for each *context* of the *TraceLink*. In line 24, rule *TraceLinkEnd2EClass* creates a new *EClass* for each *TraceLinkEnd*. In addition, the relevant to this link end maintenance attributes are added.

Listing 4.4: *Context2EClass* & *ContextData2EAttribute* Rules

```
1 rule Context2EClass
2   transform s : Trace!Context
3   to t : ECore!EClass {
4     t.eSuperTypes.add(traceContext);
5     t.name := s.name + 'Context';
6     t.eStructuralFeatures.addAll(s.data.equivalent());
7   }
8
9 rule ContextData2EAttribute
10  transform s : Trace!ContextData
11  to t : ECore!EAttribute {
12    t.name := s.name;
13    t.eType := s.type.literal.createEType();
14  }
```

4.1 Representation of traceability with TML

Rules *Context2EClass* and *ContextData2EAttribute* in Listing 4.4 transform each context and its respective data into an *EClass* and a set of *EAttributes* of the appropriate type respectively.

Listing 4.5: *MaintenanceData2EAttribute* Rule

```
1 rule MaintenanceData2EAttribute
2   transform s : Trace!MaintenanceData
3   to t : ECore!EAttribute {
4     guard : s.type.literal <> 'Set'
5     t.name := s.name;
6     t.eType := s.type.literal.createEType();
7   }
```

Listing 4.5 illustrates the rule *MaintenanceData2EAttribute*. This rule transforms all the *MaintenanceData* of a TML into a set of *EAttributes*. These attributes are added to their corresponding *link ends* and are used for the maintenance of a traceability model.

Listing 4.6: The TML2ECore Transformation expressed in ETL - *Post* Block

```
1 post {
2   traceLink.eSuperTypes.addAll(Trace!Context.all.select(c|c.`default`)
3     .equivalent());
}
```

In Listing 4.6, the *post* block of the transformation between TML and Ecore is illustrated. In ETL, the *post* block is a block of statements which are executed after the transformation rules have been executed. In the above transformation, the purpose of the *post* block is to associate all the *contexts* with the *default* attribute set to *true* with every *TraceLink*.

Transforming to EVL Constraints As discussed in Section 4.1, every traceability metamodel must be accompanied by a set of correctness constraints. In the TML approach we consider two sets of constraints. The first set consists of the constraints, which have been identified in Section 4.1.3. These constraints are common in different traceability metamodels and hence they can be abstracted into first class entities of TML and therefore they are generated automatically. The second set consists of those constraints, that are case-specific and do not apply in the general case of a traceability metamodel. These constraints can not be generated automatically from a TML model.

4.1 Representation of traceability with TML

However, in order to reduce the effort associated with the specification of traceability metamodels, we can generate a skeleton of those constraints, which then has to be filled in manually by the traceability engineers. In this section we illustrate the transformations from a TML model to the aforementioned sets of constraints and discuss their functionality.

To generate the constraints that complement the Ecore metamodel, we have employed a template-based model-to-text transformational approach using the Epsilon Generation Language (Rose *et al.*, 2008) as the template language. Again, any model-to-text language such as MOFScript (Oldevik, 2011) or XPand (Efftinge, 2006b) could have been used instead. In this section we go through the transformations and discuss their functionality. The complete transformations can be found in Appendix D.

Listing 4.7: EGL Rule for Generating the *ForAll* EVL Constraint

```
1 [%for(link in Trace!TraceLink.allInstances) { %]
2   [%for(end in link.ends.select(e|e.forAll)) { %]
3     context [%=end.type.name%] {
4       constraint OneForEach[%=end.name.firstToUpperCase()%]{
5         [%=end.computeGuard()%]
6         [%if (end.isMany()) {%]
7           check : [%=link.name%]TraceLink.all.exists(e|e.[%=end.name%].
            target.includes(self))
8         [%} else {%]
9           check : [%=link.name%]TraceLink.all.exists(e|e.[%=end.name%].
            target = self)
10        [%}%]
11        message : 'No links of type [%=link.name%] found for [%=end.name
            %] ' + self
12      }
13    }
14 [%}%]
```

Listing 4.7 demonstrates the constraint which checks the *forAll* attribute. For all *TraceLinkEnds* that have the *forAll* attribute set to true (line 2), a constraint is generated in line 4. In line 5, the guard of the constraint is generated by calling the *computeGuard()* operation which returns an appropriate guard according to the value of the *ofTypeOnly* attribute of the *TraceLinkEnd*. In lines 7 and 9, two alternative constraint bodies (*check*) are generated depending on the cardinality of the processed end. In line 11 the *message* part of the constraint - which is evaluated if the constraint is not satisfied for a particular model element - is specified.

4.1 Representation of traceability with TML

Listing 4.8: EGL Rule for Generating the *Unique* EVL Constraint

```
1 [%for (end in link.ends.select(e|e.unique)) { %]
2   context [%=end.type.name%] {
3     constraint Unique[%=end.name.firstToUpperCase()%] {
4       [%=end.computeGuard()%]
5       [%if (end.isMany()) {%]
6         check : [%=link.name%]TraceLink.all.select(e|e.[%=end.name%].
          target.includes(self)).size() < 2
7       [%} else {%]
8         check : [%=link.name%]TraceLink.all.select(e|e.[%=end.name%].
          target = self).size() < 2
9       [%}%]
10    message : 'Multiple links of type [%=link.name%] found for [%=end.
          name%] ' + self
11  }
12 }
13 [%}%]
14 [%}%]
```

In Listing 4.8, a constraint is generated based on the value of the *unique* attribute, to ensure that only one trace link is allowed for each instance of a given type. For all *TraceLinkEnds* that have the *unique* attribute set to true (line 1), a constraint is generated in line 3. As in Listing 4.7, in line 4, the guard of the constraint is generated by calling the *computeGuard()* user defined operation. In lines 7 and 9, two alternative constraint bodies (*check*) are generated depending on the cardinality of the processed end. In line 11 the *message* part of the constraint - which is evaluated if the constraint is not satisfied for a particular model element - is specified.

Listing 4.9: EGL Rule for Generating the *Optional* EVL Constraint

```
1 [%for (context in Trace!Context.allInstances) { %]
2   context [%=context.name%]Context {
3     [%for (data in context.data) { %]
4       [%=data.getInvariantType()%] NotOptional[%=data.name%] {
5         check : self.[%=data.name%].isDefined()
6         [%if (data.type.literal = 'String') {%]
7           and self.[%=data.name%].trim() <> ''
8         [%}%]
9         message : 'No value specified for attribute [%=data.name%]'
10      }
11 }
```

4.1 Representation of traceability with TML

```
11  [%}%]
12  }
13  [%}%]
```

The last generated correctness constraint is based on the value of the *optional* attribute of *ContextData* and its purpose is to ensure the non-emptiness of mandatory contextual information. The rule that generates this constraint is illustrated in Listing 4.9.

Apart from these automatically generated constraints, a traceability engineer can specify in a TML whether particular trace link types should be accompanied by particular constraints or critiques. Then TML uses EGL to generate a skeleton of those constraints, whose body has to be written manually by the traceability engineer. The rules which generate the skeleton of the constraints and the critiques are illustrated in Listing 4.10.

Listing 4.10: EGL Script for Generating the Skeleton of the Custom EVL Constraints

```
1  [%for (link in TraceLink.all.select(1|1.constraints.size() > 0)) { %]
2  context [%=link.name + "TraceLink"%] {
3  [%for (constraint in link.constraints) { %]
4    constraint [%=constraint.name%] {
5      check : true
6      message : ""
7    }
8  [%}%]
9  }
10 [%}%]
11
12 [%for (link in TraceLink.all.select(1|1.critiques.size() > 0)) { %]
13 context [%=link.name + "TraceLink"%] {
14 [%for (critique in link.critiques) { %]
15   critique [%=critique.name%] {
16     check : true
17     message : ""
18   }
19 [%}%]
20 }
21 [%}%]
```

In Lines 2 and 15 of Listing 4.10 the constraints and critiques are generated respec-

tively for the appropriate *TraceLinks*. The *check* and the *message* parts of the constraints and the critiques have to be filled in manually by the traceability engineer.

4.1.4.3 Concrete Syntax

The built-in reflective editor of the Eclipse modelling Framework (EMF) (Eclipse Foundation, 2010a) is chosen for editing and manipulating TML models. Following (Kolovos, 2007), there are four different approaches to building a concrete syntax for an Ecore-based language. In the first approach, the Graphical Modelling Framework (GMF) (Eclipse Foundation, 2010b) can be used to generate a graphical editor for the language of interest. A different approach involves the use of one of the existing model-to-text frameworks such as the Xtext framework from openArchitectureWare (Efftinge, 2006a). The third option is to use the tools, which are shipped with EMF and generate a set of plug-ins for a tree-based editor. Finally, the built-in reflective editor of Ecore can be used to generate and edit models in the XML Metadata Interchange (XMI) format.

The first three options require the generation and maintenance of additional Eclipse plug-ins, while the last one requires only the existence of the Ecore metamodel. Due to this it is considered the most flexible and the most appropriate for research activities (Kolovos, 2007). An additional advantage of the built-in reflective editor over the other three options is the fact that implementing a GMF editor, a textual concrete syntax or generating/maintaining additional plug-ins for a EMF tree-based editor is time consuming, especially for a research project, in which the metamodels change frequently. Hence, the option of the built-in reflective editor seems the most appropriate for the purposes of this work.

One main drawback of the built-in reflective editor is its lack of customisability. That is, the icons for the various model elements are identical to each other, while the various labels that represent the model elements in the editor consist of the name of the meta-class and the value of its attribute. Finally, all the models must have an *.ecore* or *.xmi* extension and not a customised extension (e.g. *.tml*), since Eclipse binds editors to file extensions.

To overcome the aforementioned issues with the built-in reflective editor, we have chosen to utilise the Extended Emf Editor (Exeed) (Kolovos, 2007). Exeed adds the customisation facilities which are missing from the built-in editor. That is, Exeed allows the developer to format labels and icons of models. An example of a TML model in an Exeed editor is illustrated in Figure 4.4. Moreover, every tree is accompanied by a properties pane in which the various properties of the different model elements can be specified. An example of the properties pane is illustrated in Figure 4.5.

4.1.4.4 Traceability Representation - Examples

In this section, we present how to define and implement semantically rich trace-links using concrete examples.

Example 1: Component and Class-Based modelling In Section 4.1.3 we have developed a traceability metamodel in Ecore (*ComponentClassTraceMetamodel*) for capturing traceability links between models that conform to two minimal *ClassMetamodel* and *ComponentMetamodel* metamodels (figure 4.2). In this section, we will attempt to reproduce the hand-crafted traceability metamodel and a subset of the associated constraints using TML instead. Our aim is to capture traceability links between instances of *Package* from the *ClassMetamodel* and *Component* from the *ComponentMetamodel*, and links between instances of *Method* from the *ClassMetamodel* and *Service* from the *ComponentMetamodel*. Furthermore, we will produce a set of related constraints directly from the TML traceability metamodel.

The first step of specifying the TML metamodel illustrated in figure 4.6 is to define the root element, i.e. the *Trace* element. In our example we call this element *ComponentClassTraceMetamodel*. Then, the two traceability links (*ComponentPackage* and *ServiceMethod*) are created. Each traceability link is associated with link ends. In the case of the *ComponentPackage* link, there are two link ends associated with it, the *package* link end of type *Package* from the *ClassMetamodel* and the *component* link end of type *Component* from the *ComponentMetamodel*.

Each link end has a set of attributes. The *package* link end has the attribute *forAll* set to true, which means that it is mandatory for all elements of type *Package* to be involved in a traceability link. In addition, the upper and the lower bound for the *package* link end are set to 1, limiting its multiplicity to 1. The attribute *ofTypeOnly* is boolean, and it is set to true. This means, that this link end can be only of type *Package* and it can

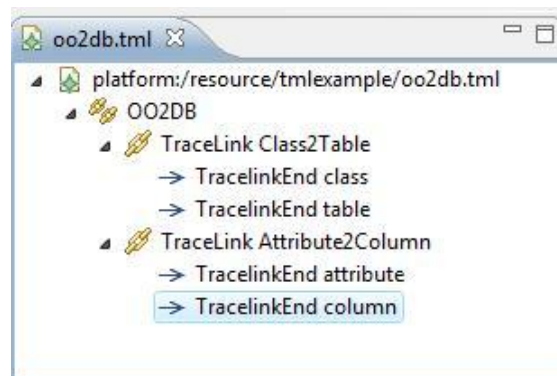
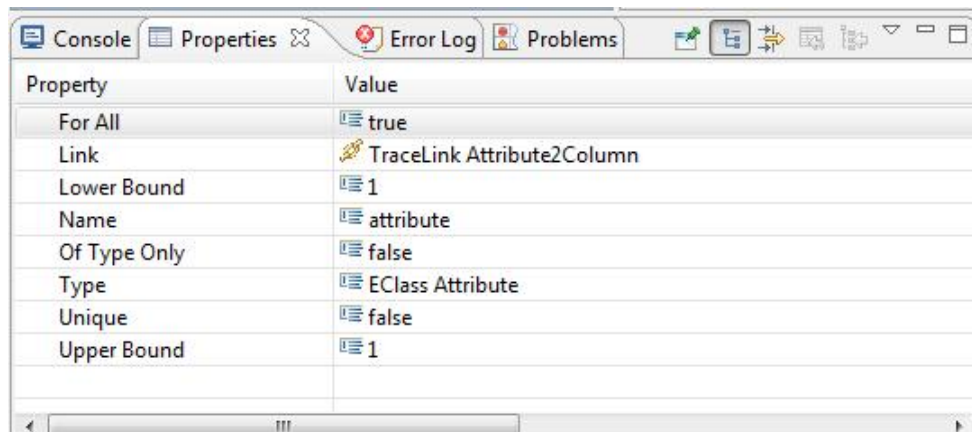


Figure 4.4: TML model in the Exeed Editor

4.1 Representation of traceability with TML



Property	Value
For All	true
Link	TraceLink Attribute2Column
Lower Bound	1
Name	attribute
Of Type Only	false
Type	EClass Attribute
Unique	false
Upper Bound	1

Figure 4.5: Example of the Properties Pane

not be of type *Model*, which extends the class *Package* in the *ClassMetamodel*. Finally, the attribute *unique*, which ensures that at most one traceability link of a particular kind exists for each instance of a given type, is set to true. This will be later interpreted as a constraint that each *component* can link to at most one *package*.

The other link end of the *ComponentPackage* link is the *component* link end of type *Component* from the *ComponentMetamodel*. The *forAll* attribute is set to true, meaning that all components must link to a package. Additionally, the upper and the lower bound for the *component* link end are set to 1, while the *ofTypeOnly* attribute's value is true. Finally, the *unique* attribute is set to true which means that each *component* can only be traced to one *package*.

Similarly, the *ServiceMethod* link is associated with two link ends, the *methods* link end of type *Method* from the *ClassMetamodel* and the *service* link end of type *Service* from the *ComponentMetamodel*. These link ends have the same set of attributes. The value of the *forAll* attribute is false for the *methods* link end, while it is true for the *service* link end. Furthermore, the multiplicity for the *methods* link end is set to be one-to-many, while the multiplicity for the *service* link end is set to be one-to-one. For both of these two link ends, the value of the attribute *ofTypeOnly* is false. Finally, the *methods* link end's *unique* attribute is set to false, whereas the same attribute for the *service* link end is set to true.

Apart from the links and their associated link ends, the TML metamodel specifies a number of contexts, which can be associated to one or more of the traceability link types. The first context specified in our example is the *Common* context, whose *default* attribute is set to true. By setting this attribute to true, we associate this context explicitly to both of the aforementioned links of our model. Each context of TML has a number of *ContextData*. In our example, the *Common* context has two, namely *description* and *author*. The former provides a short description for each traceability link, while the

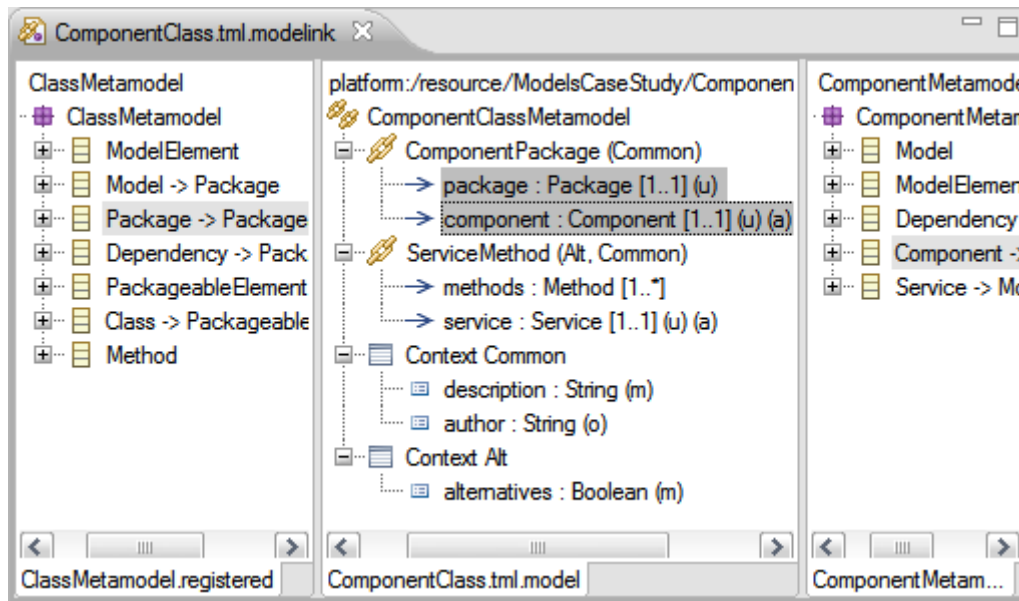


Figure 4.6: The *ComponentClassTraceMetamodel* specified using TML

latter provides the creator of each link. Each *ContextData* has three attributes associated with it, i.e. *name*, *type* and *optional*. The *description ContextData* is of type String and its *optional* attribute is set to false. This means that it is mandatory that every link in our model has a description. The *author ContextData* is of type String as well and its *optional* attribute is set to true, meaning that it is not mandatory to specify an author for each traceability link in our exemplar model.

The second context of our example is the *Alternatives* context, which is not attached to all of the trace links in the model, but only to the *ServiceMethod* traceability link. Hence, the *default* attribute of this context is set to false. Its *ContextData* is named *alternatives*, and shows if a service depends on all of the methods (logical and) or only on one of them (logical or). In our example, this is of type boolean and it is mandatory.

Generating the Ecore-based metamodel and the EVL constraints After creating a TML model, we can use the M2M and M2T transformations presented in Section 4.1.4 to obtain a traceability metamodel in Ecore and a set of associated constraints expressed in our case in EVL. The generated, case-specific traceability metamodel in Ecore is semantically equivalent to the hand-crafted one presented in Figure 4.2. In addition, the specified attributes for the various TML elements, are used to generate constraints in EVL. For example, the *forAll* attribute of the *component* link end, which is set to true in our example, produces the EVL constraint of Listing 4.11 while the *optional* attribute of the *description* context data which is set to false produces the constraint of Listing

4.12.

Listing 4.11: The EVL constraint generated by the *forAll* attribute

```
1 context Component {
2   constraint OneForEachComponent {
3     check : ComponentPackage.all.exists(e|e.component = self)
4     message : 'No links of type ComponentPackage found for component
5               '
6     + self
7   }
}
```

Listing 4.12: The EVL constraint generated by the *optional* attribute

```
1 context Common {
2   constraint NotOptionaldescription {
3     check : self.description.isDefined()
4     and self.description.trim() <> ''
5     message : 'No value specified for attribute description'
6   }
7 }
```

Example 2: Traceability in Requirements Engineering In the previous example, we demonstrated how TML can be used to specify traceability information. In this one, we will demonstrate the approach on a more realistic scenario, based on the requirements engineering (RE) domain.

In the RE literature, the RE effort is divided in two broad phases, the *early-phase* and the *late-phase* (Yu, 1997). The *early-phase* activities include those that consider how the intended system can be integrated in the organisation, how it can meet the organisational goals, why the system is needed, what alternatives exist, etc. The emphasis of this phase is on understanding the various “*whys*” that underlie the system requirements rather than on “*what*” the system should do. The knowledge obtained from the *early-phase* activities will be used to guide the activities in the later phases of the software development.

In contrast to the *early-phase* of RE, the *late-phase* focuses on coming up with a detailed requirements specification as well as on the completeness, consistency, and automated verification of this specification. Following (Yu, 1997), it is argued that because *early-phase* RE activities have different objectives and goals from those of the

the *late-phase* ones, it is appropriate to have different modelling support for the two phases.

According to Alencar *et al.* (2000), the *i** framework (Grau *et al.*, 2006; Yu, 1997) is well suited for *early-phase* requirements capture, while the Unified Modelling Language (UML) (Object Management Group, 2004) can be used for *late-phase* requirements capture. UML is not adequate to deal with early requirements capture and analysis, since it cannot describe and evaluate alternatives and their relationship to organisational objectives. Instead, the *i** framework can be used, since it allows for a better description and reasoning of the various organisational relationships among the agents of a system as well as the understanding of the rationale of the decisions taken. Hence, both UML and the *i** framework can be used complementarity to support the entire RE phase. However, since different notations are used to represent different but possibly overlapping views of the system, traceability information must be captured and maintained among the different views.

In this example, we will trace concepts between the *Strategic Dependency (SD)* metamodel of the *i** framework and a simplified version of an *OO Class Diagram metamodel*. The rationale behind the various relationships between those two metamodels can be found in (Alencar *et al.*, 2000), where a set of heuristics is provided for transforming SD models to UML class diagrams.

*i** (Grau *et al.*, 2006; Yu, 1997) is a modelling framework which focuses on the strategic actor relationships. In contrast to many standard modelling techniques like UML or Entity-Relationship diagrams, *i** provides abstractions which are capable of expressing organisational relationships among the various organisational agents and the rationale behind the various decisions taken. The various participants of the organisational setting are actors with intentional properties such as goals and beliefs. These actors depend on each other in order to fulfill their objectives and have their tasks performed. The *i** framework consists of two models: the SD model and the Strategic Rationale (SR) model.

The SD is a graph whose nodes represent the system actors and the vertices represent dependency relationships among them. The goal of an SD model is to capture the motivation and the rationale of actor's activities. An SD model distinguishes four types of dependencies; goal dependencies, task dependencies and resource dependencies. Furthermore a goal can be either a hard goal or a soft goal. In the RE literature, a soft goal is a goal, which can not be precisely defined. A soft goal is similar to a hard goal except that the criteria for whether a soft goal is achieved are not clear-cut and a priori. In a goal dependency, an actor depends on another actor for the fulfillment of one goal. In a resource dependency, an actor depends on another actor for a resource, while in a task dependency an actor depends on another actor to carry out a task. Actors can be classified into agents, roles and positions. An agent is an actor with a concrete physical existence. A role is an abstract characterisation of the behaviour of a social

4.1 Representation of traceability with TML

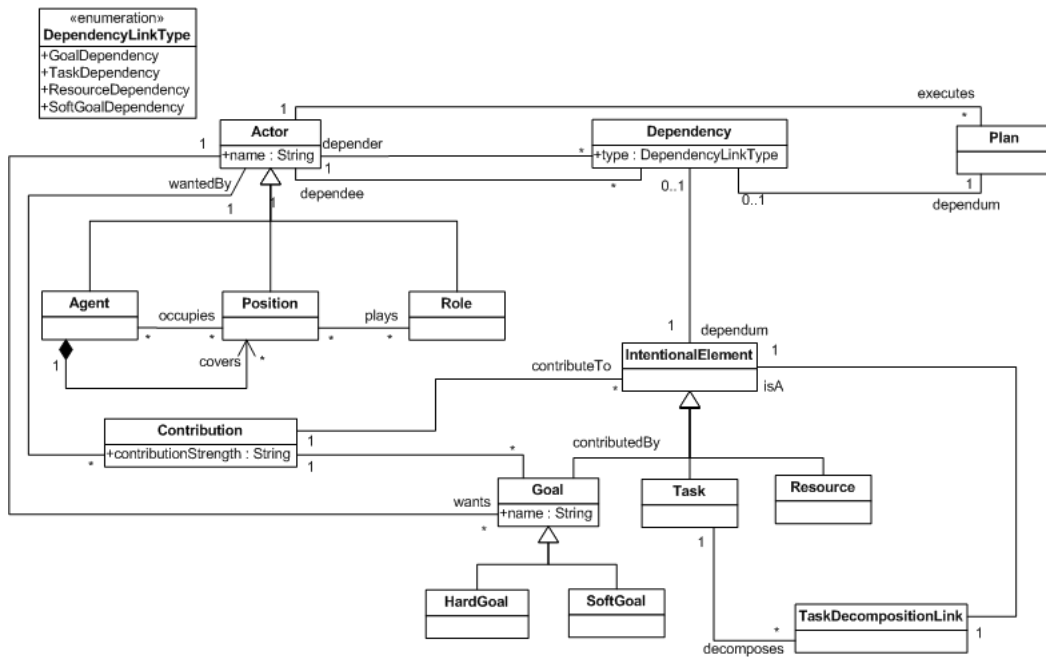


Figure 4.7: The SD Metamodel of the i* Framework

actor within some specialised context, while a position is a set of roles of a single agent. A simplified metamodel for an SD model can be found in figure 4.7.

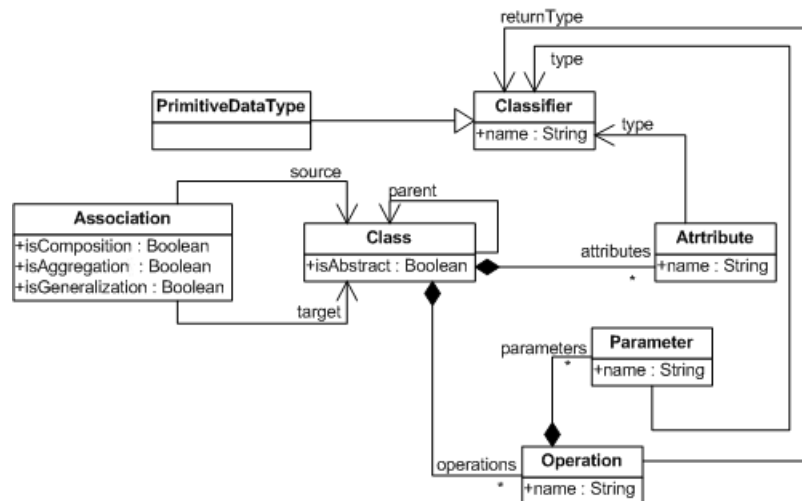


Figure 4.8: The Sample OO Metamodel

The second metamodel we will use in this example is a simplified OO class diagram metamodel. This metamodel is illustrated in figure 4.8. In the context of RE, a class

diagram can be thought of as a conceptual model of the system under consideration. It describes the various entities involved in the system, their relationships, the domain model constraints, the system scope as well as the domain-vocabulary. Hence, it provides a structural view of the system under consideration and it can be complemented by other dynamic views, such as Use Case Models Object Management Group (2004).

For this example, the following mappings will be captured:

- Instances of the *Actor* class in the i* framework will be mapped to instances of the *Class* class in the OO model.
- A *Task* in a SD model will be mapped to an *Operation* of the relevant class in the OO model.
- Instances of the *Resource* class in the i* framework will be mapped to instances of the *Class* class in the OO model.
- Instances of the *HardGoal* class and instances of the *SoftGoal* class in the i* framework will be mapped to instances of the *Attribute* class, which belong to relevant classes in the context class diagram. Since this context class diagram is part of the late-phase of RE, the recording of the various domain entities goals and their satisfaction in the form of class attributes is considered to be beneficial.

The first step of the proposed approach is to define the *IStar2OO* case-specific traceability metamodel. This metamodel specifies the *IStar2OO* class, which functions as a container for the various trace-links. Then, the five trace links (*Actor2Class*, *Task2Operation*, *Resource2Class*, *HardGoal2Attribute* and *SoftGoal2Attribute*) are created. Each traceability link is associated with the relevant link ends. In the case of the *Actor2Class* link, there are two link ends associated with it, the *Actor* link end of type *Actor* from the *IStar* metamodel and the *Class* link end of type *Class* from the *OO* metamodel.

Each link end has a set of attributes. The *Actor* link end has the attribute *forAll* set to true, which means that it is mandatory for all elements of type *Actor* to be involved in a traceability link. In addition, the upper and the lower bound for the *Actor* link end are set to 1, limiting its multiplicity to 1. The attribute *ofTypeOnly* is boolean, and it is set to false. This means, that this link end can be of any type which extends the class *Actor* in the *IStar* metamodel, such as the classes *Agent* or *Role*. Finally, the attribute *unique*, which ensures that at most one traceability link of a particular kind exists for each instance of a given type, is set to true. This will be later interpreted as a constraint that each *Actor* can link to at most one *Class*.

The other link end of the *Actor2Class* link is the *Class* link end of type *Class* from the *OO* metamodel. The *forAll* attribute is set to true, meaning that all *Classes* must link to an *Actor*. Additionally, the upper and the lower bound for the *component* link

4.1 Representation of traceability with TML

end are set to 1, while the *ofTypeOnly* attribute's value is true. Finally, the *unique* attribute is set to true which means that each *Class* can only be traced to one *Actor*. In a similar manner the rest of the trace links of the traceability metamodel are specified. The *IStar2OO* metamodel is illustrated in figure 4.9.

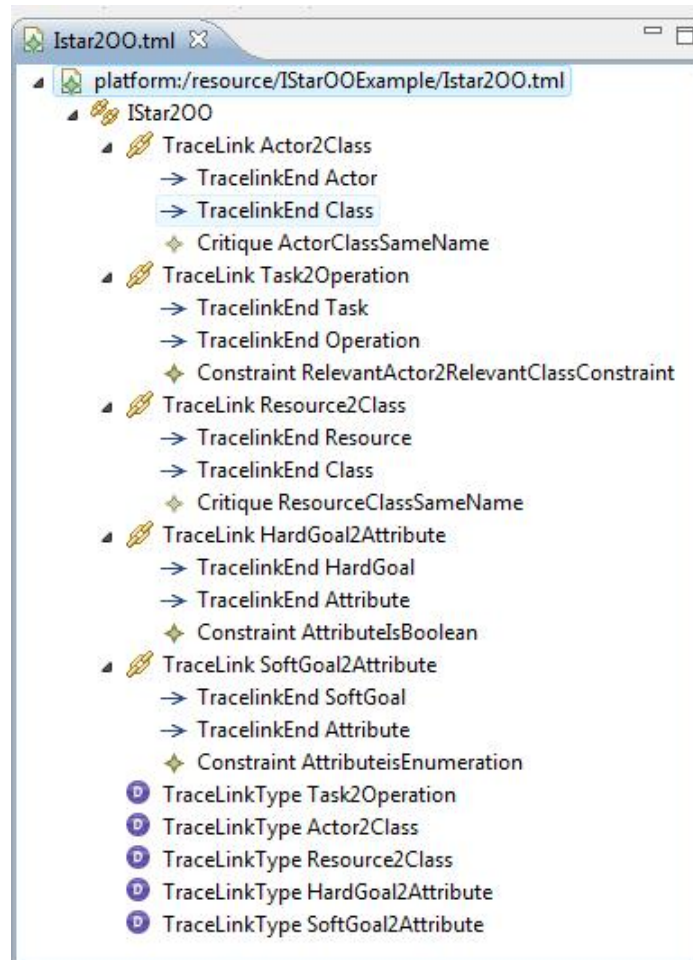


Figure 4.9: The *IStar2OO* metamodel

In the previous example, there were no custom constraints. In this example apart from the constraints, which are generated automatically, there are case-specific constraints and critiques which have to be coded by the traceability engineer. Critiques are constraints, which check for non-critical issues that should nevertheless be addressed by the user. The various constraints and critiques are modeled in the TML model as it is illustrated in figure 4.9.

4.1 Representation of traceability with TML

For the *Actor2Class* trace link the *ActorClassName* critique is specified. According to this critique, for every *Actor2Class* trace link, the name of the *Actor* reference and the name of the *Class* reference should be the same.

The *ActorClassAgreement* constraint is attached to the *Task2Operation* trace link. This constraint ensures that if a *Task* is linked to an *Operation* via a *Task2Operation* trace link, then the *Actor* to whom the *Task* belongs must be also linked to the *Class*, which contains that operation

The *Resource2Class* trace link has the *ResourceClassName* critique attached to it, which is similar to the *ActorClassName* one. This critique generates a warning if the name of the *Resource* and the *Class*, which are linked via the *Resource2Class* trace link, have a different name.

According to the heuristics provided in (Alencar *et al.*, 2000), for every *Hard-Goal2Attribute* trace link, the *Attribute* end of the link must be of type *Boolean*. This is because hard goals are well defined, hence it is always possible to establish if one has been fulfilled or not. This is captured by the *AttributeIsBoolean* constraint, attached to this trace link.

Finally, the constraint *AttributeIsEnumeration* is attached to the *SoftGoal2Attribute* trace link. This constraint ensures that for every *SoftGoalAttributeTraceLink*, the *Attribute* end of the link must be an *Enumeration*. This is due to the fact that soft goals are not well defined. They can only be *satisfied* to some degree and the different values of the *Enumeration* represent the different degrees of soft goal fulfillment.

From this set of constraints and critiques, TML can generate an EVL skeleton. An excerpt from this skeleton is illustrated in Listing 4.13

Listing 4.13: Excerpt of the EVL skeleton generated by TML

```
1 context IStar200!HardGoal2Attribute {
2   constraint AttributeIsBoolean {
3     guard :
4     check :
5     message :
6   }
7 }
8
9 context IStar200!SoftGoal2Attribute {
10  constraint AttributeIsEnumeration {
11    guard :
12    check :
13    message :
14  }
```

15 }

Once the skeleton is generated, the traceability engineer can fill in the bodies of the generated constraints and critiques. A benefit of the proposed approach is the fact that all the information required for traceability are captured in one model and they are not scattered in various artefacts. Hence, the traceability engineer can capture at a higher level of abstraction the types of constraints that are required for the various links as well as their rationale.

An excerpt of the completed custom EVL for this example is illustrated in Listing 4.14.

Listing 4.14: Excerpt of the EVL Code Created by the Engineer

```
1 context IStar200!HardGoal2Attribute {
2   constraint AttributeIsBoolean {
3     guard : self.attribute.isDefined()
4     check : self.attribute.type = OOMetamodel!PrimitiveType#'Boolean'
5     message : 'Attribute ' + self.attribute.name + ' should be of type
        Boolean'
6   }
7 }
8
9 context IStar200!SoftGoal2Attribute {
10  constraint AttributeIsEnumeration {
11    guard : self.attribute.isDefined()
12    check : self.attribute.isTypeOf(OOMetamodel!Enumeration)
13    message : 'Attribute ' + self.attribute.name + ' should be of type
        Enumeration'
14  }
15 }
```

In Listing 4.14, constraint *AttributeIsBoolean* applies to all instances of the *HardGoal2Attribute* in an *IStarOOTraceModel* trace link. In the first part of the constraint, a *guard* is defined, which checks if an attribute end is defined for the instance of the *HardGoal2Attribute* trace link. In EVL, a *guard* limits the applicability of a constraint to a narrower subset of instances of its specified type. Hence, if the guard fails, namely no attribute end is defined for the particular link instance, then the contained invariant will not be evaluated. The check part of the constraint returns true if the attribute reference of the *HardGoal2Attribute* trace link is of type *Boolean* and false otherwise. If the *check* part returns false, that is if the attribute reference of the *HardGoal2Attribute* link is not of the type *Boolean*, then an appropriate error message is produced in the message part.

Constraint *AttributeIsEnumeration* works in a similar manner. This constraint applies to all instances of the *SoftGoal2Attribute* trace link in an *IStar2OO* trace model.

As with the previous example, to establish traceability links between the SD metamodel of the *i** framework and a sample OO metamodel, we have captured all the required information in a dedicated traceability model, namely the *IStar2OO* model. From this TML we generated an Ecore metamodel in which every legitimate type of traceability link is represented as a metaclass, which contains references to the types of elements it can link. Additionally, two sets of constraints were captured. The first one consists of the constraints which can be generated automatically by the *IStar2OO* model. The second one consists of case-specific constraints which have to be written by the traceability engineer.

4.2 Trace Link Recovery with TML

In the previous section, we described how TML can facilitate the specification of semantically rich and case-specific traceability information. In this section, we will describe how this traceability information can be used to discover previously unknown trace links and to record them in dedicated traceability models.

The problem of trace link recovery is the following. Given *model* M_A which conforms to *metamodel* MM_A and *model* M_B which conforms to *metamodel* MM_B how can we automate the identification of links (relationships) between the two models? Automatic trace link recovery becomes more important when the associated artefacts are large (i.e. there are more possible links between the artefacts) as well as when the relationships between the artefacts increases in complexity. As discussed in section 2, this is an important issue in the traceability research community and there is a substantial amount of research work on this issue. In the following, we summarise the various techniques that are used in the literature for trace link recovery and we argue why these techniques are not sufficient to support this traceability activity in MDE. Furthermore, we identify a set of requirements which must be satisfied by a trace link recovery technique in the domain of MDE, and finally we present an approach which is based on TML.

4.2.1 Summarising Current Practices to Traceability Identification

In chapter 2, we presented the main methodologies to traceability identification, while in chapter 3, we listed a set of identification techniques in the form of taxonomic characters. These techniques are the following:

Techniques 1-3 come from the IR domain. These techniques are used for trace link recovery on the assumption that two related documents will share the same vocabulary

1 Identification with IR techniques - VSM	7 Identification with run-time monitoring
2 Identification with IR techniques - LSI	8 Identification augmented with A.I. Techniques
3 Identification with IR techniques - PM	9 Identification augmented with visualisation techniques
4 Identification with indirect rules	10 Identification with history analysis
5 Identification with direct rules	11 Implicit identification with transformations
6 Identification with program analysis	12 Explicit identification with transformations

Table 4.1: Traceability Identification Techniques

since developers normally choose names for the various entities from the application-domain knowledge. Moreover, the use of such techniques requires text intensive artefacts, since they are generally based on chunks of natural language text and usually do not rely on any structural properties of the input (Winkler & von Pilgrim, 2009).

Overall, trace recovery with IR techniques performs satisfactory in a situation, where the above assumptions hold. The process of recording traces is fully automatic and manual intervention is only necessary when traces are queried and used. However, the initial quality of the candidate links prior to the manual review is often poor (Winkler & von Pilgrim, 2009). As a result, the engineer who uses IR for trace recovery, has to have knowledge about a particular traceability scenario or about a particular domain, since he or she has to verify if a candidate is a correct link or not. Furthermore, these techniques do not identify all possible links, since entities can be linked even though they are not similar from a textual point of view. Finally, the links recovered by such approaches have poor semantics. Most of the IR techniques are able to retrieve pairs or sets of entities which are similar in some way, but the semantics of the links cannot be derived. That is, IR methods can identify possible links, but they do not provide further information about the nature or the type of the recovered links.

The aforementioned assumptions and concerns make IR techniques an unsatisfactory solution to traceability identification between models in MDE. In a MDE model the structural properties of the model and its textual parts are equally important, since they can capture essential information about the modeled entity. Hence, neglecting these properties for recovering trace links (as IR techniques do) limits the effectiveness of any recovery mechanism. Moreover, as mentioned in chapter 3 one of the principles of MDE is the application of system decomposition by modelling a system from a variety of viewpoints, which do not possibly share the same vocabulary (Sommerville *et al.*, 1997). Hence, relying on similar vocabularies to recover trace links between models in MDE is not ideal. Furthermore, the semantics of models and therefore the semantics of the traceability links play a crucial role in MDE as discussed in chapter 3. IR techniques though can not support rigorous semantics for the recovered links, and this limits the applicability of such techniques for this purpose in MDE. Finally, IR techniques consider only the Dependency viewpoint of traceability, while in MDE both the Dependency and the Generative viewpoints are equally important. Due to these reasons,

IR techniques are a poor candidate for automated trace link recovery between models in an MDE process.

Similar to the IR techniques, A.I. techniques are used to augment and automate approaches to trace link recovery. These techniques use mainly machine learning algorithms to categorize and find relationships between the traceable artefacts. One of the most widely used approaches is *topic modelling* (Asuncion *et al.*, 2010), which is a machine learning technique for automatically inferring semantic topics from a text corpus. Due to their nature, the A.I. techniques have the same limitations with IR techniques when it comes to traceability support in MDE. They perform better with textual than non-textual artefacts. They rely on consistent terminology in the traceable artefacts. They do not support fine-grained granularity of trace links (i.e. they usually identify links between entire artefacts and not between entities in those artefacts) and finally, they do not support rigorous trace link semantics.

The next type of trace recovery techniques consists of techniques 4 and 5. These techniques apply rules to the attributes of a traceable artefact in order to recover possible links. In the case of indirect rules, the traceability engine exploits transitivity in order to identify implicit links. For example, if a link exists from an artefact A to an artefact B and another link from B to C, then the traceability engine is able to automatically derive and record a link from A to C. On the other hand, in the case of direct rules, assertion rules are defined to identify similarity between two traceable entities. For example, the rule can exploit lexical analysis techniques to identify similarity between the entities based on common occurrences of a keyword.

Most rule-based approaches require some form of human interaction in order to set up a basis for traceability. This may be either a set of traces recorded manually, or a basis of customized rules. Compared to recording traces completely manually, the costs are lower. However, flexibility is also generally decreased, as most approaches rely on aspects such as particularly structured artifacts or consistent use of terminology. Moreover, no guidance or assistance is provided to the engineer to create the various rules. As a result, writing complex rules can be error prone and time consuming.

Techniques 6 and 7 rely on program analysis to recover previously unknown trace links. The former technique relies on static program analysis techniques to identify links between software entities, while the latter relies on dynamic program analysis techniques for the same purpose. Although, these techniques can be efficient for recovering links between software entities, this is not the case for recovering links between MDE models. First, in most cases models in MDE are not executable, hence dynamic program analysis does not apply. Second, many of the links that exist between models are informal and they can not be recovered by this kind of analysis. Relevant to this is the fact that the techniques, which rely on program analysis for trace link recovery, can not capture the semantics of trace links, since they can identify only a limited number of link types. Therefore, from an MDE perspective, such approaches are not ideal for

trace link recovery between models.

History analysis is another technique used to recover trace links. This technique utilizes version control systems such as subversion (Pilato, 2004) to identify change patterns among sets of files that are modified together frequently in the past. The underlying assumption of this technique is that the change patterns can be used to recommend potentially relevant documents. The main shortcoming of this approach is that although frequent common modifications might imply some kind of a relationship, they can not inform the engineers on the type of this relationship. Therefore, the semantics of the trace links can not be defined using a history analysis approach.

The next type of trace link recovery method makes use of visualisation techniques to augment the manual declaration of trace links. Such a technique can be for example a special coloring schemes for particular types of artefacts, when they are displayed in a tool. The user can make use of the visual information to manually establish trace links. Since, the establishment of the links is done manually, the cost of traceability remains high, especially for large or complex artefacts. Moreover, such approaches do not provide any support for defining the semantics of the trace links in an enforceable way.

The final type of trace link recovery techniques consists of techniques 11 and 12. These techniques rely on transformations in order to identify and establish possible trace links between artefacts. In the case of recovering trace links implicitly, the required traceability information can be generated by the traceability engine, which is used to execute the transformation. As a by-product of the transformation the transformation engine generates traceability information in the form of the following tuple:

`<source element, transformation rule, target element>`

The source is the element which is used as source in the transformation, the target is the resulting element and the third element of the tuple is the transformation rule, which was used to generate the target element from the source element. This traceability information do not provide any particular link types or semantics for the different types of traceability relationships that can exist.

In the second case of trace link recovery using transformations, links can be established explicitly by injecting traceability-related code in the transformation code. This approach has the advantage, that links with particular semantics can be created. On the other hand though, the transformation code becomes “polluted” from the traceability-related code, hence it becomes less readable. Finally, both explicit and implicit trace recovery techniques neglect completely the Dependency viewpoint of traceability and they focus solely on the Generative viewpoint.

From the above summary of the various trace recovery methods, it can be seen that none of the above methods was built having in mind the principles and special conditions which apply to an MDE process as described in section 3.4. In the following

section, we identify the requirements of a trace link recovery method for MDE, while in section 4.2.3, we present a novel approach to trace link recovery based on the TML.

4.2.2 Requirements for Traceability Identification

The purpose of this section is to summarise the characteristics a trace link recovery method in MDE should have based on the discussion which took place in the previous sections. These characteristics are:

1. **Model-centric approach:** In order to provide MDE-specific support, a trace link recovery method should utilize both the structural metadata as well as the lexical information of a model element in order to recover possible trace links. This is due to the fact that the main artefacts in MDE are models, and for such graph-like artefacts as models the structural properties of an element are equally important to its lexical parts, since they can capture essential information about the modeled entity.
2. **Support for rich traceability semantics:** Since the semantics of trace links is very important as discussed in section 2.4.2, a trace link recovery technique should be able to recover links with rich semantics, and not just a generic relationship between two entities.
3. **Support for both Dependency and Generative Identification:** Since initial and derived models are used in MDE and they are very often related to each other, a trace recovery technique for MDE should be able to support both Generative and Dependency viewpoints. That is, such a technique should be able to recover trace links between a generated model and the model which was used for this generation, as well as trace links between models which do not share this kind of relationship, but co-exist.
4. **Automated Recovery:** Trace link recovery is considered a costly and error prone activity (Egyed *et al.*, 2005). Hence automating this activity is of paramount importance.
5. **Framework-agnostic:** Since in MDE there is a plethora of model management frameworks (Mens *et al.*, 2005), a trace link recovery method should be generic and not tied to any particular technology.

4.2.3 Identification in TML

In the previous section, we presented the main trace link recovery methods and we identified a set of requirements, which should be satisfied by such a method in the context of

MDE. From this discussion it is apparent, that the main issue of the existing approaches is the fact that the recovered links do not have rich, scenario-specific semantics. TML does not propose a new recovery technique but its purpose is to enhance the links recovered by model management frameworks with case-specific semantics. Moreover, the proposed approach is able to support both the Dependency and Generative viewpoints of traceability. This is a very important feature in the context of an MDE process, since both initial and derived models are used in such processes.

An MDE process typically consists of a number of model management (e.g. model to model transformation, model validation) and classical software development (e.g. code compilation, file system management) tasks (Kolovos, 2008b). Many of these tasks generate an internal trace. For example, an ETL module provides the *exportTransformationTrace* attribute or an Epsilon Comparison Language (ECL) (Kolovos, 2009) module provides the *exportMatchTrace* attribute that enable the developer to export the internal trace to the project context. TML relies on such model management languages for a set of traceability links. In the case where a model management language does not provide such an internal trace, the TML approach proposes the injection of traceability-related code in the code written in the model management language in order to generate the required set of trace links. An example of this case is provided in section 6, where EOL is used to describe a transformation instead of ETL.

Trace link recovery in TML relies on two model-to-model transformations. The first transformation is from the internal trace generated by a model management task to a generic trace representation, while the second transformation is from the generic trace representation to the case-specific traceability model.

Many model management frameworks are currently used to manipulate models (Kolovos, 2008a). As TML should not be restricted to a particular model management framework or to a particular language, which supports particular model management tasks, a key purpose of the generic trace representation is to provide an abstraction layer through which model management frameworks will be able to uniformly utilise the functionality offered by TML. The metamodel of this internal representation is shown in figure 4.10.

The *GTrace* class acts as the root of a generic trace model and it can contain a number of links, which are captured by the *GTraceLink* concept. Finally, each link can have two or more traced elements. Each of the main three concepts of a generic trace model (i.e. *GTrace*, *GTraceLink* and *GTracedElement*) can be annotated. The details of these annotations have the form of key-value mappings and their purpose is to capture useful details for the model management task of interest. For example, an annotation can capture which of the link ends is the source of a transformation and which one is the target.

In the following sections, we will present these transformations, as well as how TML supports the Generative and Dependency viewpoints.

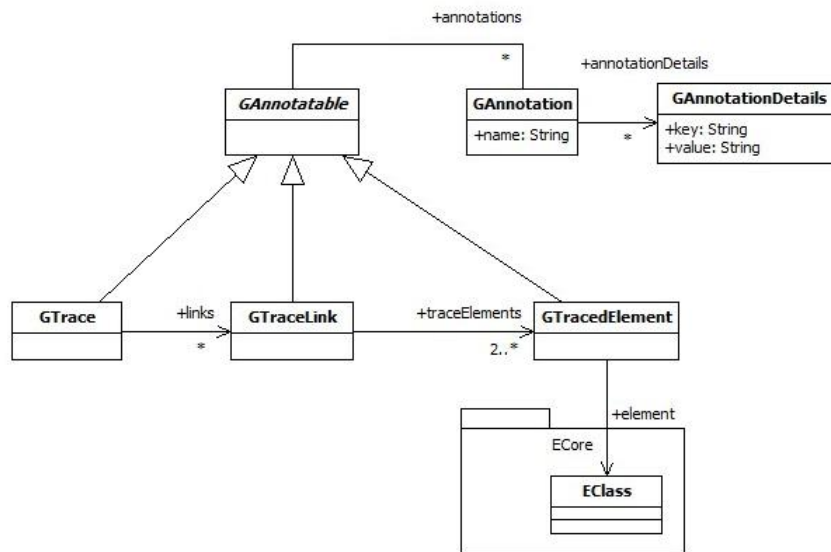


Figure 4.10: Generic Trace Metamodel

4.2.3.1 Internal Trace to Generic Trace

The first transformation of this transformation-chain is the one from the internal trace of a model management task to a generic trace representation. Since, the purpose of the TML approach is to support both the Dependency and Generative viewpoints, we have currently implemented two transformations, which cover these cases. To support the Generative Viewpoint, we have implemented a transformation, which takes as a source the internal trace produced by ETL and produces a generic trace. To support the Dependency Viewpoint, we have implemented a transformation, which takes as a source the internal trace produced by ECL and produces a generic trace. These two transformations, as well as their rationale and how they can support the two viewpoints, are discussed in the following.

Generative Viewpoint Due to the importance of model transformations in MDE, the support of the Generative viewpoint is of paramount importance for any traceability approach whose focus is on an MDE process. Many declarative and hybrid transformation languages such as ETL and ATL have built-in support for traceability. Such support is needed to link elements generated by different transformation rules together. For example, a target element may be referred to by using its corresponding source element as a key. Such a form of traceability however does not have rich semantics and usually does not persist after executing a transformation. For this reason, it is called internal traceability (Jouault, 2005).

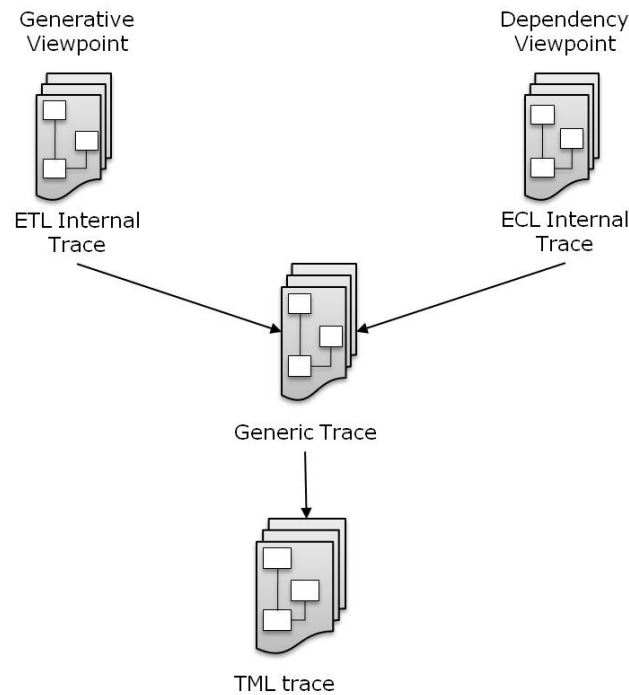


Figure 4.11: Transformation-Chain

In the TML approach, we use this internal traceability to generate a trace represented in an intermediate generic format and then using a TML model and this generic representation, we generate case specific traceability models. For this thesis, we have used the internal trace generated by the ETL, but in principle any transformation engine which generates internal traces could have been used. In the case where there is no internal trace, then the generic trace can be generated by injecting traceability-specific code in the transformation code. In this section we will first present how the proposed approach works in the case, where an internal transformation trace is available, and then we will discuss how traceability can be established when the transformation does not produce such a trace. In the case of a transformation language, which generates an internal trace, the algorithm for generating the generic trace model works in the way described by pseudocode in algorithm 1.

When a transformation is executed a *GTrace* model is created. For every link of the internal trace model of the transformation engine a *GenericTraceLink* is created and an *GAnnotation* is attached to it. The name of this annotation is the transformation language used. For example, in the implementation for this work this will be “ETL”. Moreover, the key of the *GAnnotationDetails* is “rule”, while its value is the name of the transformation rule, from which this link is created. Each link of the internal trace has a set of source elements and a set of target elements depending on the trans-

Transformation 1 Internal Trace to Generic Trace - Transformation Language

```

create a GTrace model
for all links in internalTrace do
  create a GenericTraceLink
  create and attach a GAnnotation to GenericTraceLink
5:  set GAnnotation.name ← " ETL"
  set GAnnotationDetails.key ← "rule"
  set GAnnotationDetails.value ← name of transformation rule
  for all source elements in links do
    create GTracedElement
10:  set GTracedElement ← element
    attach a GAnnotation
    set GAnnotation.name ← " ETL"
    set GAnnotationDetails.key ← "kind"
    set GAnnotationDetails.value ← "source"
15:  end for
  for all target elements in links do
    create GTracedElement
    set GTracedElement ← element
    attach a GAnnotation
20:  set GAnnotation.name ← " ETL"
    set GAnnotationDetails.key ← "kind"
    set GAnnotationDetails.value ← "target"
  end for
end for

```

formation language. The next step of the algorithm is to take all the source elements of the internal link and create the relevant *GTracedElements*. Moreover a *GAnnotation* is attached to every *GTracedElement*, whose key is set to “kind” and its value to “source”. The purpose of this annotation is to capture the directionality of the link, by specifying that a particular *GTracedElement* of a *GTraceLink* is a source. Similarly, for every target element of the internal trace link the algorithm generates the corresponding *GTracedElements* with an *GAnnotation*, whose key is set to “kind” and its value to “target”.

In the case of a language which does not support an internal trace, traceability re-

lated code has to be injected in the transformation code in order to generate the *GTrace* model. The form and content of this code depend on the transformation language we are targeting. For this work, we have implemented traceability support for the EOL, an OCL-based imperative model management language built on top of the Epsilon platform (Kolovos & Paige, 2010). To this end, we have implemented the following class in Java, which has the *add* operation. This operation takes as parameters a source element, a target element and a transformation rule and adds a trace link to an internal trace.

Listing 4.15: Java implementation of the *Trace* class

```
1 public class Trace extends TransformationTrace {
2     public void add(Object source, Object target, String ruleName) {
3         ArrayList<Object> targets = new ArrayList<Object>();
4         targets.add(target);
5         super.add(source, targets, null);
6     }
7 }
```

When we want to generate a trace in an EOL script, we just have to invoke the *add* method directly from within the EOL code. This is illustrated in listing 4.16.

Listing 4.16: Invoking the *add* method in EOL

```
1 // TML: Trace Source to Target
2 trace.add(source, target, "");
```

In section 6, the injection of traceability code in an EOL script is demonstrated in a case study in which EOL is used to implement a complex (≈ 1000 loc) model-to-model transformation.

Dependency Viewpoint The second viewpoint supported by the TML approach is the Dependency viewpoint. In this case, traceability must be established between existing models, which do not have the kind of relationship between a source and a target of a model transformation. Such traceability has to capture some implicit or explicit semantic relation, or dependencies, between the related models. The majority of the traceability approaches we have presented in the previous sections cover this kind of traceability. However, most are not developed for recovering traceability between models. Given the structured nature of models traditional text-based comparison approaches have been shown to be insufficient for finding similarities between models (Treude *et al.*, 2007).

In this work we propose the use of model comparison in order to discover semantic and structural similarities between models. Once such similarities have been discovered and an internal trace is created, the semantics of this internal trace is enhanced in a

similar way with the way described in the previous section, where it is described the traceability recovery support in the Generative case.

Model comparison is the task of identifying matching elements between models. In general, matching elements are elements that are involved in a relationship of interest (Kolovos, 2009). Several approaches to model matching have been proposed in the literature. Id-based approaches establish matches between models based on the common non-volatile identity of model elements (e.g. (Alanen & Porres, 2003)). However, these approaches cannot be applied to modelling technologies that do not support such identifiers or to individually developed models. Signature-based approaches overcome the limitations of id-based approaches by the identities used for the comparisons are computed automatically using the features of each element (Fleurey *et al.*, 2007). The main issue with this approach to model comparison is the fact that string-based identifiers are not meaningful for all types of model elements. The third type of approaches to model comparison consists of similarity-based approaches (e.g. (Treude *et al.*, 2007)). These approaches treat models as typed attribute graphs and calculate the similarity between nodes based on the combined weighted similarity of their features. The main limitation of the similarity-based approaches is that they are generic and they can not be easily configured to exploit the semantics and particularities of the various modelling languages. Moreover, they can not be applied to heterogeneous models, which conform to different metamodels.

For this work, we have chosen to use the ECL, which is a hybrid, task-specific, model comparison language. ECL overcomes the aforementioned limitations of the similarity-based approaches by enabling developers to specify precise language-specific algorithms for establishing matches between elements conforming to different metamodels in a rule-based manner. ECL produces an internal trace similar to the internal trace produced by transformation languages like ETL. More precisely, the internal trace of ECL has the following form:

<source, target, matching rule >

In the case of a comparison language, which generates an internal trace, the algorithm for generating the generic trace model is similar to the algorithm used for a transformation language. The transformation is presented using the following pseudocode. When a model comparison is executed a *GTrace* model is created. For every link of the internal trace model of the comparison engine a *GenericTraceLink* is created and an *GAnnotation* is attached to it. The name of this annotation is the comparison language used. For example, in the implementation for this work this will be “ECL”. Moreover, the key of the *GAnnotationDetails* is “rule”, while its value is the name of the comparison rule, from which this link is created. Each link of the internal trace has a source element and a target element. The next step of the algorithm is to take the source element of the internal trace link and create the relevant *GTracedElement*. Moreover a *GAnnotation* is attached to every *GTracedElement*, whose key is set to “kind” and its value to

Transformation 2 Internal Trace to Generic Trace - Comparison Language

```

create a GTrace model
for all links in internalTrace do
  create a GenericTraceLink
  create and attach a GAnnotation to GenericTraceLink
5:  set GAnnotation.name ← " ECL"
  set GAnnotationDetails.key ← "rule"
  set GAnnotationDetails.value ← name of comparison rule
  for all source elements in links do
    create GTracedElement
10:  set GTracedElement ← element
    attach a GAnnotation
    set GAnnotation.name ← " ECL"
    set GAnnotationDetails.key ← "kind"
    set GAnnotationDetails.value ← "source"
15:  end for
  for all target elements in links do
    create GTracedElement
    set GTracedElement ← element
    attach a GAnnotation
20:  set GAnnotation.name ← " ECL"
    set GAnnotationDetails.key ← "kind"
    set GAnnotationDetails.value ← "target"
  end for
end for

```

“source”. The purpose of this annotation is to capture the directionality of the link, by specifying that a particular *GTracedElement* of a *GTraceLink* is a source. Similarly, for every target element of the internal trace link the algorithm generates the corresponding *GTracedElements* with an *GAnnotation*, whose key is set to “kind” and its value to “target”.

In the case of a comparison language which does not support an internal trace, traceability related code has to be injected in the comparison code in order to generate the *GTrace* model. The form and content of this code depend on the comparison language we are targeting. This part of the approach is identical to the one described for the

Generative viewpoint.

4.2.3.2 From a Generic Trace Model to a Traceability Model

The second step of the transformation-chain consists of the transformation from a *GTrace* model to the traceability model. This transformation enhances the information contained in the *GTrace* model with case-specific semantics provided by the *TML* model. The resulting traceability model conforms to the traceability metamodel generated by the transformation of the case-specific *TML* model to the case-specific traceability metamodel expressed in Ecore, which is described in section 4.1.4.2. This transformation is described using the pseudocode in algorithm 3.

Usually, executing a model management task requires the declared models and metamodels to be bound to actual models in the file system. This is called the *operational context* of the transformation (Jouault *et al.*, 2006). This is usually done in the launch configuration of the task. In the *TML* approach, apart from the models and metamodels, the *TML* model which corresponds to a particular task belongs to its operational context. In our implementation, in which we provide *TML* support for ETL and ECL, the traceability model is passed as a parameter to the execution engine. An example of this is illustrated in figure 4.12, where it is shown the launch configuration pane of an ETL transformation. By extending the operational context, the postprocessor of the model management language, has all the necessary information for transforming the generic trace to a traceability model.

This transformation creates first a trace model for the model management task it corresponds to, if such model does not exist. The next step in the transformation is to iterate across all the links in the *GTrace* model and try to match them with suitable links in the *TML* model, which is part of the operational context of the model management task. If more than one suitable links are found then, the *GTraceLink* is flagged as ambiguous and a warning is generated stating the alternatives for this link. If no suitable link is found then the *GTraceLink* is flagged as invalid and a warning is generated. An invalid link means that there is no suitable trace link in the *TML* model. An assumption we make is that a meaningful relationship between two elements should be captured in the traceability model. Hence, when a *GTraceLink* is invalid, this indicates that there might be an error in the transformation rule which created this link. We will expand further into this aspect of *TML* in section 4.4, where we describe how a traceability model can be used to test the correctness of particular aspects of a transformation. The last part of this transformation is executed if only one suitable link is found for the *GTraceLink* under consideration. If such a link is found then a link is created in the trace model and its link ends are created with their corresponding reference values. The type and name of this link are set to be the same with the ones of the link found in the *TML* model and its status is set to be *OK*. Moreover, the maintenance data of the link are generated according to the information found in the *TML* model. Once the trace

Transformation 3 Internal Trace to Generic Trace - Comparison Language

```

if TraceModel is not defined then
    create a TraceModel
end if
for all GTraceLinks in GTrace do
5:   find suitable links among the TraceLinks in TMLModel
    if number of suitable links > 1 then
        flag the link as ambiguous
        produce user warning
    else if number of suitable links = 0 then
10:   flag the link as invalid
        produce user warning
    else
        create Link in TraceModel
        Link.name ← GTraceLink.name + "TraceLink"
15:   Link.type ← TraceLink.type
        for all TraceLinkEnds in TraceLink do
            create LinkEnds of Link
            add addReferenceValues to LinkEnds
            add TraceLinkEndOKStatus
20:   add MaintenanceData to LinkEnds
        end for
    end if
    validate TraceModel with EVL constraints
end for

```

model is generated, the corresponding correctness constraints which accompany the TML model are executed to ensure the correctness of the generated traceability model. If the validation process returns errors and assuming that the TML is specified correctly, this is an indication that the model management task has errors.

To find suitable links to a *GTraceLink*, the model management task postprocessor, attempts to match a *GTraceLink* with a link specified in the TML model. To do this, the postprocessor attempts to match the link end types and the cardinalities of the *GTraceLink* with the link end types and the cardinalities of the links in the TML model. If a match is found, then it is assumed that the *GTraceLink* corresponds to that link.

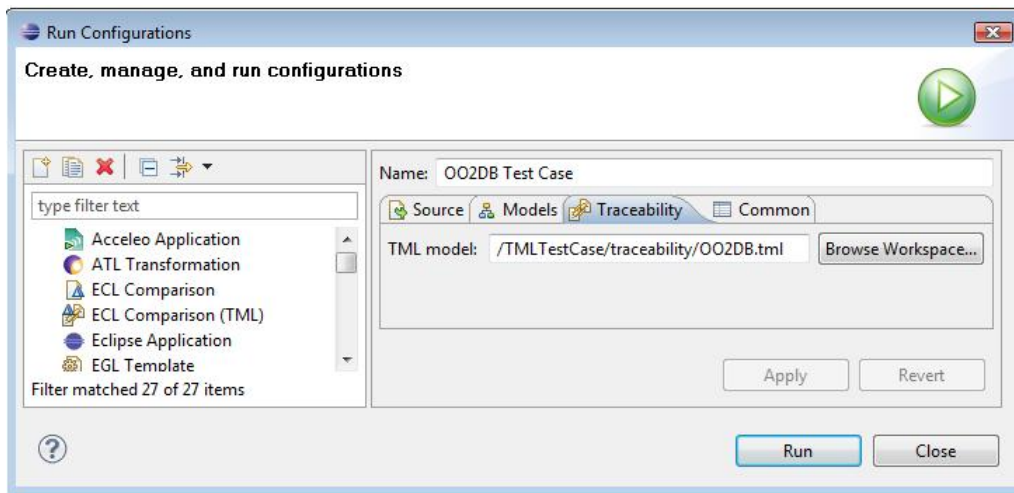


Figure 4.12: ETL launch configuration

In the case, where more than one candidate links exist, the postprocessor needs extra information in order to decide between the different alternatives. This information is provided to the postprocessor in the form of annotations, which are added the rule or operation which created a particular link in the internal trace and specify to which trace link type the internal link corresponds. The name of such annotations is *TML* and their value is a trace link type. Their concrete syntax is illustrated in listing 4.17.

Listing 4.17: TML annotation example

```
1 @TML traceLinkType="linkType"
```

In the case that traceability code is injected in a model management task code (that is if the model management language does not produce an internal trace) then the signature of the *add* operation, which is described in listing 4.15, is changed so that it takes a particular link type as parameter. Adding TML annotations to a model management tasks means that the developer of such tasks is familiar with the TML model which defines the traceability between the various model elements. This is an assumption of the proposed approach. We argue that this assumption is valid, since the TML model should describe all the valid relationships between models and all the subsequent actions which create instances of such relationships should do this in a way which is consistent with the TML model.

For example, consider the case where model elements e_A and e_B , which belong to models M_A and M_B respectively, are related in two different ways. The semantics of the first relationship is captured by trace link $[e_A - to - e_B]_{generatedBy}$ and this link

captures the fact that e_B is generated by e_A using an ETL transformation. The semantics of the second relationship is captured by trace link $[e_A - to - e_B]_{rationalisation}$ and this link captures a rationalisation relationship between the two elements as defined by (Spanoudakis & Zisman, 2005). When the ETL transformation is executed, the post-processor will match the generated *GTraceLink* of the rule which transforms e_A to e_B to both $[e_A - to - e_B]_{generatedBy}$ and $[e_A - to - e_B]_{rationalisation}$ links in the TML model, which defines the traceability between the metamodels to which the models of interest belong. This is because both the link ends of those links as well as their cardinalities are identical. Due to this, the transformation rule will be flagged accordingly to inform the user that it created an ambiguous link. To avoid this the developer of the transformation can add the annotation illustrated in listing 4.18 to the transformation rule in order to inform the ETL postprocessor which is the trace link in the TML model which corresponds to this transformation rule.

Listing 4.18: TML annotation example

```
1 @TMLtraceLinkType="eA2eBGeneratedByTraceLink"  
2 rule eA2eB  
3   transform s : MMa!eA  
4   to t : MMb!eB {  
5     ...  
6   }
```

At this point it should be noted that the proposed approach cannot generate the various *Contexts*, which might be attached to the trace links in the TML model. These contexts often store informal information about a trace link, hence their automatic generation is not possible. The developer has to add them manually once the traceability model is generated. This is why, the proposed approach is characterised as semi-automatic and not as fully automatic. To our view though, automatic recovery of trace links with rich, scenario-specific semantics is difficult to achieve due to the informal nature of traceability information. Following (Pineiro, 2004), this informality is unavoidable, since a number of software engineering activities are inherently informal, such as the initial stages of requirements elicitation.

4.2.4 Traceability Recovery with TML - Conclusions

This section has provided a detailed discussion on traceability recovery using TML. TML proposes the use of existing model management operations for generating trace links between models and the enhancement of the generated traces with rich semantics using a transformation chain and a TML model. Initially, the internal trace, which is

generated by the model management operation is transform to a generic trace. Since, TML does not target a particular model management framework, an abstract representation is required to hide the differences and particularities of the different frameworks, which can be supported by TML. This abstraction layer is provided by the *GTrace* notation, which is a generic traceability representation. The second step of the transformation chain consists of a transformation from a *GTrace* model to a case-specific traceability model.

In section 4.2.2, a set of requirements has been identified for a traceability recovery approach in the context of MDE. The proposed approach in this thesis satisfies the identified requirements is in the following way:

1. **Model-centric approach:** since the proposed approach relies on existing model management frameworks in order to recover a set of trace links, it is considered to be model-centric. Model management frameworks provide operations, which are used on models and they produce internal traces.
2. **Support for rich traceability semantics:** the proposed approach utilises the traces produced by model management operations and enhances them with scenario-specific semantics, which are captured in a TML model.
3. **Support for both Dependency and Generative identification:** the proposed transformation utilises the internal trace produced by model transformations in order to provide support for the Generative viewpoint. On the other hand, the proposed approach utilises model comparison for supporting the Dependency viewpoint. This is achieved by finding structural and semantic similarities between co-existing models using an appropriate model comparison language, whose internal trace is used for generating the semantically-rich trace links.
4. **Automated recovery:** to a great extent, the proposed approach automates the recovery of trace links. Informal aspects of the trace links though have to added manually to the traceability model.
5. **Framework-agnostic:** Due to its design, the proposed approach is not restricted to a particular model management framework. The current implementation provides transformations for EOL, ETL, and ECL, but TML could provide traceability support for other model management operations and frameworks. This can be done by implementing appropriate transformations.

In the next section, we will present an example on how trace links can be recovered using TML.

4.2.5 Traceability Identification - Example

In this section, we give a concise example of the proposed approach to trace link recovery. The purpose of this example is to demonstrate how trace links can be recovered between models which conform to the two metamodels defined in section 4.1.3. In our example, we want to recover trace links between a class model, which conforms to the *ClassMetamodel* and models the booking system of a restaurant, and a component model, which conforms to the *ComponentMetamodel* and which is generated by the *ClassModel* using a model-to-model transformation written in ETL. An EMF tree representation of the class model of the restaurant booking system is shown in figure 4.13.

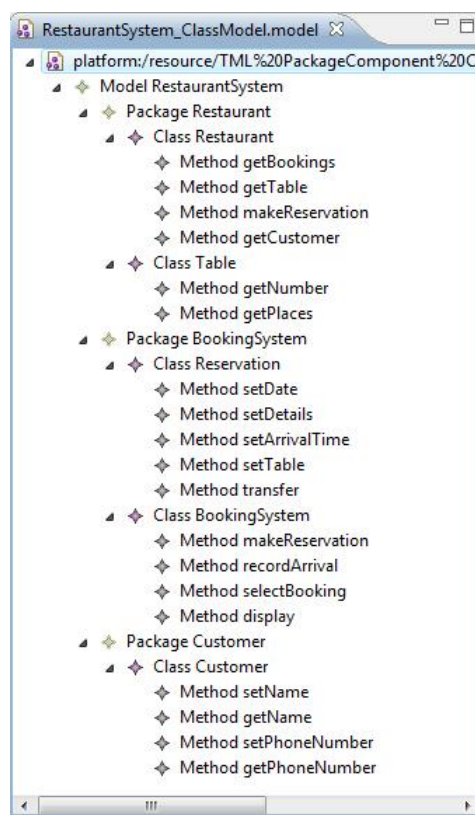


Figure 4.13: Restaurant Booking System Class Model

This model consists of three packages, namely the *Restaurant* package, the *BookingSystem* package and the *Customer* package. The *Restaurant* package models entities related directly to the restaurant, hence it contains the *Restaurant* and *Table* classes. These classes provide a set of methods such as *getBookings*, *getTables*, *getNumbers*, etc. The second package is the *BookingSystem* package and it consists of the *Reser-*

vation and *BookingSystem* classes. Finally, the last package is the *Customer* package, which contains only one class, namely the *Customer* class.

This class diagram can be transformed to its corresponding component diagram. This scenario often arises when a simple analysis class diagram is transformed to a component diagram to describe the architectural design of the system. This transformation written in ETL is illustrated in listing 4.19. The *Package2Component* rule transforms each *Package* to a *Component* and it assigns to it the name of the *Package* as well as the relevant services which correspond to the methods of the *Classes* which are contained in the *Package*. The second rule, which is called *Method2Service*, transforms each method of a *Class* in the *ClassModel* to a *Service*.

Listing 4.19: Class-2-Component ETL transformation

```
1 rule Package2Component
2   transform s : ClassModel!Package
3   to t : ComponentModel!Component {
4     t.name = s.name;
5     for(class in s.contents){
6       for (service in class.methods.equivalent()){
7         t.services.add(service);
8       }
9     }
10  }
11 rule Method2Service
12   transform s : ClassModel!Method
13   to t : ComponentModel!Service {
14     t.name = s.name;
15  }
```

In a typical ETL transformation scenario, the developer would pass the two models as the operational context of the transformation and the transformation would have been executed. In the case of the proposed approach, apart from the source and target models the developer has to include the TML model to the operational context of the transformation, so that the post-processor can use the case-specific traceability information in order to recover trace links with rich semantics. The TML model for this scenario is the one defined in section 4.1.4.4. Figure 4.14 illustrates the altered ETL run configuration, in which the TML model is passed as a parameter to the postprocessor of the transformation.

When the above transformation is executed the component model for the restaurant booking system is generated, but at the same time the traceability model is gener-

4.2 Trace Link Recovery with TML

ated with all the *Package2ComponentTraceLink* and *Method2ServiceTraceLink* links, as specified in the TML model. Moreover, the associated constraints with the TML model are run by the postprocessor to verify that the generated traceability model is correct. A snapshot of the generated traceability model is shown in figure 4.15.

In this example, we did not need to put any annotations in the transformation code, since the links specified in the TML model referred to different model elements. Therefore no ambiguity could arise in resolving the links during the generation of the traceability model.

In the example so far, we have demonstrated how the proposed approach can be used when the component model is generated automatically from the class model. In the scenario where the component model is derived manually from the class model (Dependency viewpoint), the approach is very similar. The only difference is that instead of using a transformation, we use a model comparison script to identify similarities between the two models and generate an internal trace. The component model for this scenario is shown in figure 4.16.

Such a model comparison script written in ECL is illustrated in listing 4.20. This script consists of two rules. The first one compares the *Packages* of the class model with the *Components* in the component model. If a *Package* has the same name with a *Component* and the *Methods* of its contained *Classes* have the same name with the *Services* of the *Component*, then a *match-trace* is added to the internal trace of the comparison. The second rule compares all the *Methods* in the class model with all

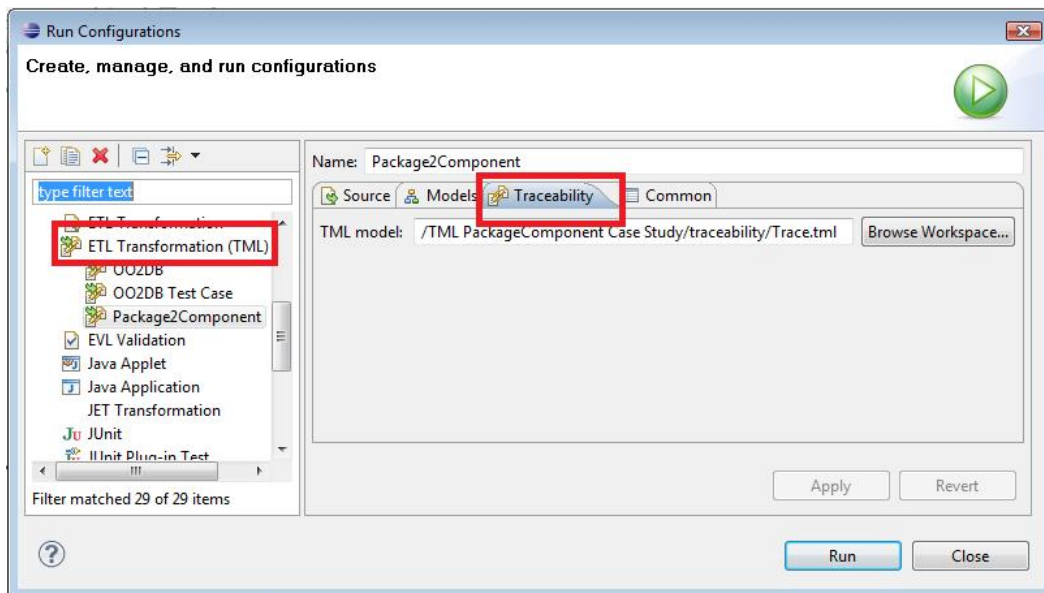


Figure 4.14: ETL with TML run configuration

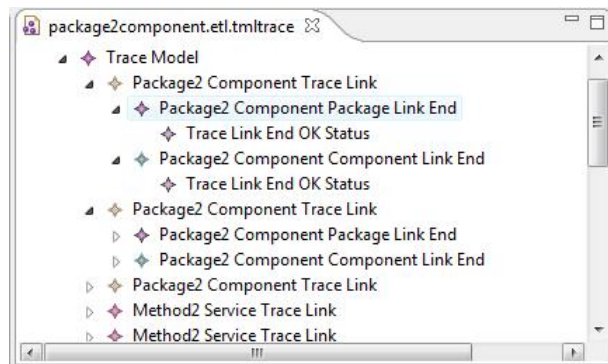


Figure 4.15: Restaurant Booking System Component Model

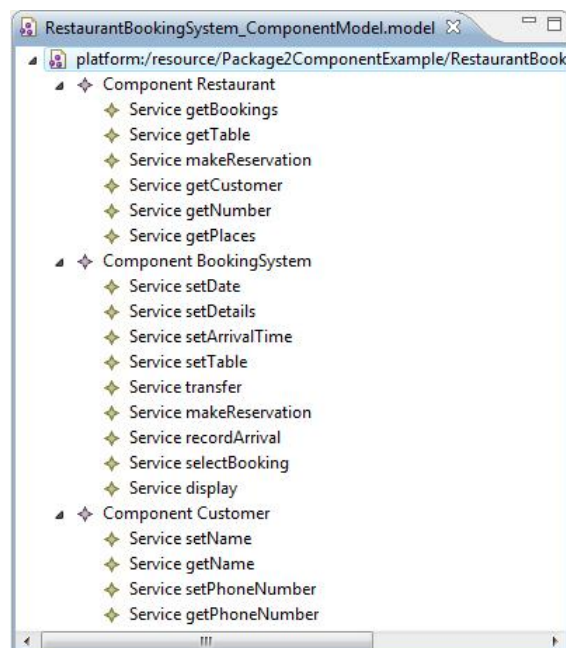


Figure 4.16: Snapshot of the generated traceability model

the *Services* in the component model. A match-trace is added to the internal trace of the comparison, if their names match. The final part of the script consists of a user-defined operation, which is applied to the context of a *Package* and returns true if a string signature, which consists of the names of the *Methods* of the *Classes* contained in it, equal a string signature of the names of the *Services* of a *Component*, which is passed as a parameter to the operation.

Listing 4.20: Class-2-Component ECL comparison

```
1
2 rule PackageComponent
3   match l : ClassModel!Package
4   with r : ComponentModel!Component {
5     compare : l.name = r.name and l.matchMethods2Services(r)
6   }
7
8 rule Method2Service
9   match l : ClassModel!Method
10  with r : ComponentModel!Service {
11    compare : l.name = r.name
12  }
13
14 operation ClassModel!Package matchMethods2Services(component :
    ComponentModel!Component) : Boolean{
15   var t : String;
16   for(class in self.contents.select(c|c.isTypeOf(Class))){
17     t = t+class.methods.collect(c|c.name).concat(',').toLowerCase();
18     if (hasMore){
19       t=t+', ';
20     }
21   }
22   if(t.equals(component.services.collect(s|s.name).concat(',').
    toLowerCase)){
23     return true;
24   }
25   else{
26     return false;
27   }
28 }
```

The trace links in this case are recovered in a similar way to the approach followed in the case of the Generative viewpoint. The quality of the links recovered in this case depends on how well the comparison script can identify true similarities between the two models. User input during the execution of the comparison can enhance its results. This facility is provided by ECL, and this is one reason we have chosen ECL for the reference implementation.

4.3 Maintenance of traceability with TML

In the previous section, we described how trace links with rich semantics can be recovered using a TML model. In this section, we will deal with the problem of traceability maintenance. One of the most challenging aspects of traceability is how to maintain the integrity of the relationships, i.e. trace links, while the referenced entities continue to change and evolve. The problem of traceability maintenance can be described informally using the following example. Consider the case where traceability model T_A , which conforms to traceability metamodel TM_A , captures traceability between models M_A and M_B which conform to metamodels MM_A and MM_B respectively. If one of the models M_A or M_B (or even both) change, the validity of the trace links in the traceability model may be affected. The activity of restoring the validity of model T_A is called traceability maintenance. Since in most traceability scenarios, a lot of time and effort is required to be invested for the specification and capture of the traceability information, traceability maintenance is of paramount importance in order to avoid degradation of the captured traceability information.

Understanding how link integrity can be violated and how it can be restored requires the examination of both the trace links and the changes on the models, which are referenced by the links. In the next two sections, namely sections 4.3.1 and 4.3.2, we analyse those two parameters of the traceability maintenance activity.

4.3.1 Link Types

(Aizenbud-Reshef *et al.*, 2006a) argue that the way a relationship is managed depends on the type of the relationship. More precisely, they identify two types of relationships or trace links. An *imposed* relationship is a relationship between entities that exists by volition of the relationship creator and it can not be invalid until its creator determines that it is invalid. For example a *satisfies* relationship can be created between a requirement and an element of a design model. When the requirement changes it is not clear if the relationship is still valid or not.

On the other hand an *inferred* relationship is one that exists because the related entities satisfy a rule that describes the relationship. An inferred relationship can not be invalid since if the rule is not satisfied the relationship does not exist. An example of an inferred relationship is the relationship which is a result of a model transformation.

4.3.2 Model Change Types

Knowing just the type of a link is not enough from a maintenance point of view. To understand how traceability maintenance works one also needs to know the types of model changes, which take place at the trace link ends. For instance, in the case of an

imposed relationship, a link can be invalidated without input from its creator, if one of the entities, which are referenced by the link ends, is moved or deleted.

There are three atomic model change types - addition, modification or deletion. In the literature, many additional types of change have been identified such as model element replacement or model merging (for example in (Mäder *et al.*, 2009)). In the context of this work, we consider all other types of change as a combination of the three aforementioned atomic types. For example, a model element replacement consists of two different actions, element deletion and element addition. Similarly, model merging can be considered as the compound effect of deleting two models and adding a new one, which is the union of the two.

Addition of models or model elements can not cause a link to be *invalid*. However, it affects the completeness of the link set; namely, new models or model elements may be added and all the appropriate links to and from the new entities are not added. Deletion can result to link integrity errors since a link end can possibly point to an entity which no longer exists. Finally, modification can occur at two different levels. First, whole models can be modified. This includes the moving of the model from one directory to another or the renaming of the model. Second, internal modifications can take place at the model element level. Such modifications include the change of element attributes in a way that the validity of the link is compromised. For example, in a model transformation scenario, if we change the type of an element in the source model, then the trace link with the relative element in the target model might not be valid any more.

4.3.3 Summarising Current Practices to Traceability Maintenance

As discussed in section 2, there are two main approaches for maintaining trace link integrity, namely event-driven and state-based approaches. The main limitation of event-driven approaches has to do with their limited scope. Limiting the scope of such an approach means limiting its applicability to particular notations or to particular tools in a way, which can not be easily generalised to include additional notations or tools. The reason this limitations are introduced is because of the three challenges to identifying compound change events. These challenges are the following (Mäder *et al.*, 2008):

- The same development activity can be achieved by performing different elementary changes.
- The same development activity can be achieved by the same elementary changes performed in different sequences
- The type of a model change and the impacted model element do not offer enough information for relating changes to each other

Because of the aforesaid limited scope of event-driven approaches, they do not provide the flexibility envisaged by the proposed approach, since TML targets diverse modelling languages which possibly conform to heterogeneous metamodels and are manipulated in different tools.

The second *family* of approaches to traceability maintenance consists of the state-based approaches. State-based approaches are often limited in detecting only syntactic model changes, while semantic model changes can jeopardise the link integrity as well. To our knowledge, an approach which addresses this issue does not exist up till now.

4.3.4 Requirements for Traceability Maintenance

If we consider the key characteristics of link integrity and traceability maintenance as discussed above, we can derive a set of requirements for a traceability maintenance approach.

1. **(R1) Detection and Correction of Invalid Links:** a traceability maintenance approach should be able to not only detect dangling links (i.e. links whose integrity is compromised), but it should also be able to re-establish the link integrity. In the ideal case, the approach should be able to reestablish the compromised links automatically, i.e. without any input from the user. In the cases where this is not possible (for example with some types of imposed links), the approach should give guidance to the user as to how to reconcile the link. This could be achieved by reducing the space of possible link end candidates. In the worst case scenario, when guidance for the reestablishment of the link integrity can not be provided, then all dangling links should at least be reported back to the user for manual reconciliation. The purpose of a good traceability maintenance approach should be to maximize the automation of the reconciliation activity without ignoring the semantics of the reconciled links.
2. **(R2) Support for both imposed and inferred link maintenance:** As discussed in section 4.3.1, there are two generic types of links. A traceability maintenance approach should be able to manage both of those link types.
3. **(R3) Support for different model change types:** In section 4.3.2, three atomic model change types have been identified. A traceability maintenance approach should be able to detect dangling links caused by all three types, namely deletion, addition and modification. A note should be made at this point concerning the addition model change type. If addition is not a part of a chain of changes then it does not compromise link integrity. However it compromises the completeness of the link set and hence an appropriate trace link identification approach should be applied. In the case where addition is a part of a chain of changes (for example

a model element replacement) then the traceability maintenance approach should be able to detect possible dangling links caused by this chain of changes.

4. **(R4) Support for heterogeneous notations:** A traceability maintenance approach should not restrict its scope to a particular notation or to a particular way models can be evolved. It should be able to maintain link integrity between models represented in heterogeneous notations no matter how these models change.

Based on this set of requirements, we have developed a traceability maintenance approach in the context of TML. This approach is illustrated in the next section.

4.3.5 State-Based Traceability Maintenance in TML

In figure 4.3 the TML metamodel is illustrated. In this metamodel, a set of concepts related to the maintenance of traceability is defined. The classes which correspond to those concepts are the following:

- the *MaintenanceData* class
- the *ReconciliationExpression* class
- the *FuzzyMatchingTechnique* class
- the *MaintenanceDataType* enumeration
- the *FuzzyMatchingImplementation* interface

In a TML model, every *TraceLinkEnd* can have 0 to n maintenance data. By maintenance data, we mean data which represent particular characteristics of the model element to which a link end refers. For example maintenance data could be a simple name attribute or more complex data structures as we will demonstrate in the next section. The rationale behind capturing maintenance data is that there are some aspects of an element which justify the existence of a link. If these change then the validity of the link comes under question. These aspects are domain specific and the domain expert can capture them in a TML model. Every instance of the *MaintenanceData* attribute of a link end is associated with a reconciliation expression, which is an expression in a model query language and it can access the associated data of the model element. Initially it is used to assign a value to the *MaintenanceData* attribute and then it is used during the traceability maintenance activity in order to retrieve the actual model element data, so that it can be compared with the value of the *MaintenanceData* attribute of the link end.

The first action which is performed during the maintenance of a traceability model is the identification of links whose integrity might be compromised. This is achieved

by checking first if there are links in the traceability model, which point to no existent elements. This can be the case when a model element is deleted for example. Moreover, the maintenance engine compares the *MaintenanceData* attributes of every link end in the traceability model with the actual data of the various model elements to which a link refers. As said above, the actual data are retrieved using the relevant to the *MaintenanceData ReconciliationExpression*. If those two are identical, this means that the link is valid. If not, the integrity of the link is compromised and appropriate action must be taken in order to reconcile the link. When a link is found to be dangling, the traceability maintenance engine attempts to reconcile it. If this is not possible, then the user is notified of the broken link.

To reconcile a broken link, the traceability maintenance engine traverses all the model elements in order to detect one whose actual data, which are derived using the reconciliation expression, match the maintenance data of the link end. A link end can have more than one *MaintenanceData* attributes associated to it. Moreover, each of those attributes can have one or more *FuzzyMatchingTechniques*. These *FuzzyMatchingTechniques* are user-defined algorithms, which are used for the comparison of the value of a *MaintenanceData* attribute with the data which are derived from the model element using the *ReconciliationExpression*. This functionality is provided to the end user, because there are often cases when a less-strict approach to matching the two values is desired. For instance, when a model element is renamed to an ontologically equivalent name (e.g. Client and Customer). In this case, to achieve a more precise matching, the use of a lexical database (e.g. WordNet (Miller, 1995)) is necessary. Alternatively, fuzzy string matching algorithms such as those presented in (Chapman, 2005) can be used. Each *FuzzyMatchingTechnique* has two attributes, namely *upperThreshold* and *lowerThreshold*. If the result of the matching is above the value of the *upperThreshold* attribute, then we treat this match as an exact match. If the result of the matching is between the value of the *upperThreshold* and the value of the *lowerThreshold*, then this match is treated as an approximate match and user input might be required. Finally, if the result of the matching is below the value of the *lowerThreshold*, then we assume that no matching element was found using the *MaintenanceData*. The algorithm which is used to identify candidate links for a *MaintenanceData* attribute is described using the pseudocode in algorithm 4.

First, this algorithm checks whether there are fuzzy matching techniques associated to a *MaintenanceData* attribute. In the case where no such techniques are defined there are two alternatives. If the actual data derived from the model element of interest match the maintenance data, then a candidate is identified since we have an exact match. In the case that there is no such match, the engine is not able to detect a candidate model element using the given maintenance data. If there are fuzzy matching techniques associated to the *MaintenanceData* attribute, then the algorithm uses them in order to compare the actual data with the maintenance data. In this scenario, the best result re-

Algorithm 4 Traceability Maintenance Algorithm - Fuzzy Matching Techniques

```

def matchingResult
  if TraceLinkEnd has no FuzzyMatchingTechniques then
    if actualData=maintenanceData then
      matchingResult  $\leftarrow$  exactMatch
5:   else
      matchingResult  $\leftarrow$  noMatch
    end if
  else
    matchingResult  $\leftarrow$  noMatch
10:  for all FuzzyMatchingTechniques of TraceLinkEnd do
      compare actualData with maintenanceData using
      FuzzyMatchingTechnique
      if result > upperThreshold then
        matchingResult  $\leftarrow$  exactMatch
      else if result < upperThreshold and result > lowerThreshold then
15:      if matchingResult  $\neq$  exactMatch then
        matchingResult  $\leftarrow$  approximateMatch
      end if
      end if
    end for
20: end if

```

turned is the one used. For example, consider the scenario where the *MaintenanceData* attribute of a given link end has three fuzzy matching techniques associated to it. The result returned the first one for a particular model element is that the *MaintenanceData* attribute matches exactly the actual data which are derived from this model element. The result returned using the second fuzzy matching technique is that the *MaintenanceData* attribute partially matches the actual data which are derived from this model element. Finally, the result returned using the third fuzzy matching technique is that there is no match. In this case, the result returned by the first fuzzy matching technique will be used, since it is the best one. This means that the model element under consideration can be used to reconcile the link end.

In addition to having many fuzzy matching techniques attached to a particular *MaintenanceData* attribute, there can be multiple *MaintenanceData* attributes attached to a given link end. During reconciliation, these attributes can produce different results. For

example, such a case is when a link end has two *MaintenanceData* attributes associated to it and during the reconciliation process, the use of the first attribute returns that a given model element is a candidate for this link end, while the use of the second one returns that this model element should not be considered as a possible candidate for the link end. The algorithm for deciding whether a model element is a possible candidate for a dangling link end or not is illustrated using the pseudocode 5.

Algorithm 5 Traceability Maintenance Algorithm - Multiple Maintenance Data per Link End

```

def overallMatchingResult
  for all MaintenanceData of a LinkEnd do
    matchingResult ← match(maintenanceData, actualData)
    if matchingResult=approximateMatch then
5:     if overallMatchingResult≠exactMatch then
        overallMatchingResult ← approximateMatch
      end if
    else if matchingResult=noMatch then
      if overallMatchingResult=null then
10:        overallMatchingResult ← noMatch
      end if
    else if matchingResult=exactMatch then
        overallMatchingResult ← exactMatch
    end if
15: end for

```

At the beginning of the search for possible candidates for a broken link end, a variable called *overallMatch* is initialised with a *null* value. For each of the *MaintenanceData* attributes of the link end a match operation is performed in order to compare the value of the attribute to the actual data derived from the various model elements and the *overallMatch* attribute is assigned values accordingly. A *noMatch* value means that a model element should not be considered as a possible candidate, while an *approximateMatch* means that there is partial similarity and user input is needed. Finally, an *exactMatch* value means that the *MaintenanceData* attribute matches the values derived from the given model element. The decision table for deciding the overall match result (i.e. using all the available maintenance data) is shown in table 4.2.

The last step of maintaining a broken link using the proposed approach is to create a reference between a dangling link end and a candidate model element, if such an element is found, or otherwise report the link as broken back to the user, so that the link

Table 4.2: Decision table for overall matching result for a given link end

matchingResult	overallResult		
	noMatch	approximateMatch	exactMatch
noMatch	noMatch	approximateMatch	exactMatch
approximateMatch	approximateMatch	approximateMatch	exactMatch
exactMatch	exactMatch	exactMatch	exactMatch

can be reconciled manually. To do so two lists are created for every broken link end. The contents of those lists are the results of the matching tasks described above. The first list contains all the model elements, which are found to be *exact matches* using the various *MaintenanceData* attributes. If this list has only one element then a reference is created between the link end and this model element and the *MaintenanceData* attributes of the link end is updated based on its new target. If this list has more than one elements, then user input is required in order to choose which of the elements is the most appropriate for the link end under consideration. Finally if this list is empty the second list is checked. The second list consists of the model elements, which are found to be *approximate matches* using the various *MaintenanceData* attributes. In the case where there are elements in this list user input is requested in order to reconcile the link. Finally, if this list is empty, the link is reported as broken to the user, who performs the maintenance, and manual reconciliation is required in order to re-establish the integrity of the broken link.

Traceability maintenance can run either manually when the traceability engineer wishes to reestablish the integrity of the traceability model or it could run automatically when model changes are detected. In the current implementation the first case is supported. However in future work the second option can be supported by coupling the proposed approach with indexing tools *Concordance* Rose *et al.* (2010). *Concordance* provides a cross-model reference reconciliation client, which can detect changes incurred at the model level.

4.3.6 Maintenance of Traceability with TML - Conclusions

This section has provided a detailed discussion on traceability maintenance using TML. The proposed approach proposes the use of dedicated maintenance meta-data and accompanying expressions with which those data can be derived for maintaining broken links. The proposed approach can be enhanced with the use of fuzzy matching techniques in order to match link ends with candidate model elements.

The proposed approach to traceability maintenance satisfies the requirements iden-

tified in section 4.3.4 in the following way:

1. **(R1) Detection and correction of broken links:** the approach is capable of not only detecting but also of reconciling broken links. The success of the approach relies on how well and precisely the traceability engineer captures the required maintenance data for a given link end.
2. **(R2) Support link maintenance for both imposed and inferred links:** Since the rationale for a link is captured in the TML model by the model engineer, the approach does not rely on an explicit relationship between model elements for reconciling the trace links. Therefore, it supports both imposed and inferred link maintenance.
3. **(R3) Support link maintenance for all model change types:** The proposed approach does not rely on identifying model changes, since it is a state-based approach. Therefore, different types of model changes do not affect the applicability of the approach directly.
4. **(R4) Support for heterogeneous notations:** TML is a generic approach to traceability. All the required traceability information is captured in the TML model from which the maintenance script is generated. Therefore, for every set of notations a TML model can be defined. The only constraint of TML is that the metamodels of the various notations should be expressed in a common metamodel. In the current implementation this is Ecore, which is the metamodel of the Eclipse modelling Framework Eclipse Foundation (2010a).

In the following, we will present how a traceability model can be evolved and how trace link integrity can be maintained using a simple example.

4.3.7 Traceability Maintenance - Example

In this section, we present an example whose purpose is to demonstrate how we can maintain the integrity of trace links between models which conform to the two metamodels defined in section 4.1.4.4. These two models model the booking system of a restaurant, and they are the same with the ones used in section 4.2.5. Figure 4.17 shows the *Class* and *Component* models for this example.

The TML model for this scenario is illustrated in figure 4.18. This TML model is the same as the one used in the example in section 4.2.5. The only difference is that the TML model for this example is enhanced with dedicated maintenance data. The first step of the proposed approach to traceability maintenance is to capture maintenance data for the link ends of the trace links we want to maintain. For demonstration purposes in this example, we will maintain the integrity of the trace link between instances

4.3 Maintenance of traceability with TML

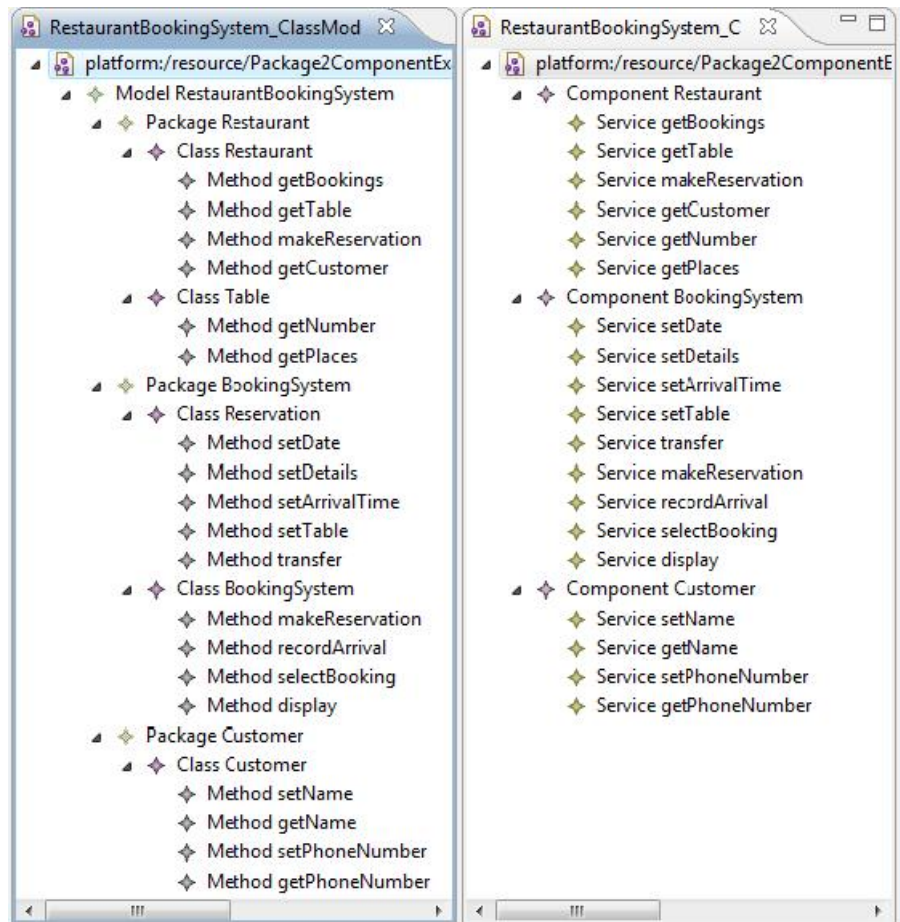


Figure 4.17: Class and Component Models for the Restaurant Example

of the *Package* meta-class and instances of the *Component* meta-class. We assume that the validity of such links can be jeopardised if the name of the package is changed, if the name of the component is changed or finally if the classes contained in a package change. One could argue at this point that there is no need for a dedicated approach to traceability maintenance in this scenario, since a developer can re-execute the transformation after changing the models of interest and a valid traceability model can be regenerated. Doing so has two main consequences. First, if the traceability model has additional informal information such as various contexts attached to the different trace links, this will be lost in the case of the re-generation of the traceability model. Second, in the case of the dependency viewpoint of traceability, models co-exist and one is not generated from the other using a transformation. Due to these reasons, regenerating a traceability model whenever a change occurs in the referenced models is not considered as a solution to the problem of traceability maintenance.

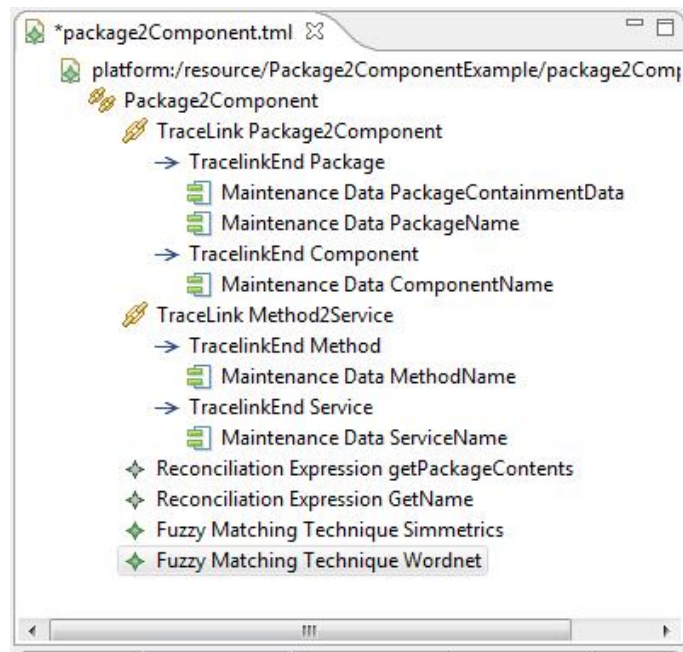


Figure 4.18: Class-to-Component TML Model

To come back to the provided example, we have attached two *MaintenanceData* attributes to the *Package* meta-class. The first one is called *PackageContainmentData* and it is a string signature of the names of the various classes contained in a package. The reconciliation expression associated to this attribute is captured in the *ReconciliationExpression* model element named *getPackageContents*. The expression captured by it is shown in listing 4.21. This expression can be applied to the context of a *Package*. It selects all elements contained in a *Package* class, which are of type *Class*, and creates a String signature with the concatenation of the names of the classes contained in a package. The properties of this *ReconciliationExpression* is illustrated in figure 4.19.

Listing 4.21: GetPackageContents Reconciliation Expression

```
1 self . contents . select (c|c.isTypeOf(Class)). collect (c|c.name).concat(',')
```

A *FuzzyMatchingTechnique* is attached to the *getPackageContents*. This technique uses the SimMetrics open source extensible library of similarity metrics (Chapman, 2005). SimMetrics provides a library of float based similarity measures between string data. The implementation of the SimMetrics tool used in this example is illustrated in listing 4.22. This code compares two strings using the Levenshtein algorithm (Gusfield, 2007). Moreover, in the properties view of the *FuzzyMatchingTechnique*, we have

4.3 Maintenance of traceability with TML

specified an upper threshold of 0.8 and a lower threshold of 0.5. The values for these attributes depend on the accuracy required by the traceability engineer. Setting very high upper and lower thresholds means that the two compared strings have to be extremely similar in order to return a match. In this case, the traceability engineer wishes to have full control over the maintenance process and he or she wants to make sure that the automatic reconciliation returns only valid results. This is done of course on the expense of less automation, since user interference is required more often.

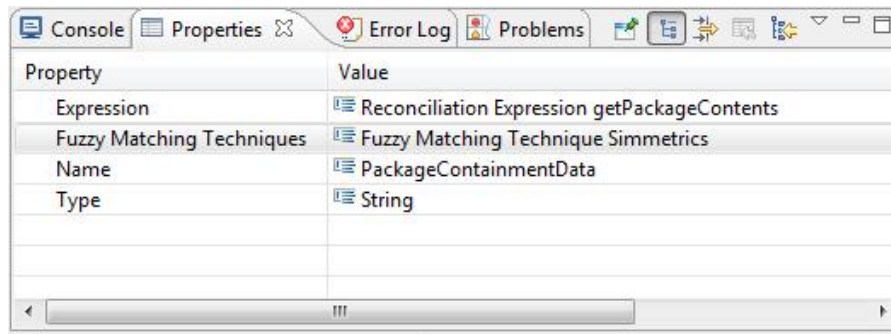


Figure 4.19: Properties of the *getPackageContents ReconciliationExpression*

Listing 4.22: Java implementation of fuzzy matching tool

```
1 package org.eclipse.epsilon.tml.fuzzymatching.simmetrics;
2
3 import org.eclipse.epsilon.eol.exceptions.EolInternalException;
4 import org.eclipse.epsilon.eol.exceptions.EolRuntimeException;
5 import TML.impl.FuzzyMatchingImplementationImpl;
6
7 public class SimmetricsTool extends FuzzyMatchingImplementationImpl{
8     public double similarity(String s1, String s2)
9         throws EolRuntimeException {
10         uk.ac.shef.wit.simmetrics.similaritymetrics.
11             AbstractStringMetric metric;
12         if (s1 == null || s2 == null) {
13             if (s1 == null && s2 == null) {
14                 return 1.0;
15             } else {
16                 return 0.0;
17             }
18         }
19     }
20 }
```

```
19     metric = (uk.ac.shef.wit.simmetrics.similaritymetrics.  
                AbstractStringMetric) Class.forName("uk.ac.shef.wit.  
                simmetrics.similaritymetrics.Levenshtein").newInstance();  
20     return metric.getSimilarity(s1, s2);  
21 } catch (Exception e) {  
22     throw new EolInternalException(e);  
23 }  
24 }  
25 }
```

Apart from the *getPackageContents* attribute, we have attached to all four link ends *MaintenanceData* attributes associated with the name of the model element they refer to. More particularly, we have attached the *PackageName* attribute to the *Package* link end, the *ComponentName* attribute to the *Component* link end, the *MethodName* attribute to the *Method* link end and the *ServiceName* attribute to the *Service* link end. All of these attributes are related to the *getName* expression. This simple expression (*self.name*) returns the name of a model element a trace link end refers to. For the *PackageName*, *ComponentName*, *MethodName* and *ServiceName* attributes, we have used the WordNet and the SimMetrics tools for supporting fuzzy matching. As it can be seen by this example complex expressions can be used as reconciliation expressions and their complexity is only limited by the expressiveness of the query language that is used. Furthermore, the *MaintenanceData* attributes can capture both structural as well as lexical properties of the referenced models.

Once the *MaintenanceData* attributes have been specified in the TML model, as well as the appropriate *ReconciliationExpressions* and *FuzzyMatchingTechniques*, the models of interest can be altered and the traceability information between them can be adjusted accordingly.

4.4 Traceability usage in the context of MDE

In the previous sections, we described how to define traceability models with rich, case-specific semantics using TML. Moreover, we have presented how trace links between models can be automatically recovered and how the integrity of the recovered links can be maintained when the referenced models change. Such traceability models can be used during the development life cycle of a software system in multiple ways to support different activities. Different traceability usage scenarios are summarised in section 2.4.4. Most of the scenarios in the literature are generic in the sense that they are not directly related to a development paradigm. In this section, we will present two traceability usage scenarios which are directly related to using traceability information in the context of MDE.

The first scenario has to do with the use of traceability information in order to test model transformations. In the existing literature, it is suggested that traceability relationships can be used to check for the existence of appropriate test cases for testing different requirements. In this work, we propose that traceability information can be also used to test the correctness of transformation or translation activities, such as a model-to-model transformations. This traceability usage scenario is described in section 4.4.1.

The next traceability usage scenario has to do with model refactorings. A model refactoring is considered to be an action that automatically creates, updates or deletes a model element. This action can be considered as an update transformation, whose source and target models are the same. This type of model transformation is usually needed when a small subset of a model needs to be changed. It is common though that when a model element changes, there are model elements in other models that need to be changed as well. For example using the working example of this thesis, if the *Package* in a class model is renamed, then the corresponding *Component* element in the component model needs to be renamed accordingly. In this work we propose the use of traceability information in conjunction with in-place transformations in order to refactor a model model and propagate the changes to the related models. This traceability usage scenario is presented in section 4.4.2.

4.4.1 Transformation Validation with TML

One of the main claims of MDE is that increased productivity can be achieved by providing automated support for creating and transforming software models. Effective support for model transformations is thus of paramount importance for the successful realization of MDE in practice. Such support should include efficient techniques and tools for testing the correctness of model-to-model transformation specifications. Testing a transformation means determining whether it has been correctly carried out (Harrold, 2000). The motivation for testing transformation specifications is described in (Jing, 2007).

Following (Harrold, 2000), transformation testing involves the “execution of a deterministic transformation specification with test data (i.e., input to test cases) and a comparison of the actual results (i.e., the target model) with the expected output (i.e., the expected model), which must satisfy the intent of the transformation”. If the actual target and the expected models match, then it can be assumed that the transformation is correct given the initial input. On the other hand, if the actual target and the expected models do not match, this means that the transformation specification is incorrect and that it needs modification. (Harrold, 2000) argues that the comparison of the actual transformation target with the expected output can be a challenge to transformation testing. Manual comparison of models can be error-prone and time consuming, while the automated comparison using graph matching algorithms can be expensive in terms

of computation requirements for such a task. Moreover, another intrinsic limitation of this *black box testing* approach to model transformations, is the fact that testers usually have to manage large input and output models due to the need to test the entire transformation specification as a whole (Ciancone *et al.*, 2010). Considering the transformation as a whole can be problematic especially for large models, since more effort is required for the design of the tests as well as for their execution.

To address these limitations (Ciancone *et al.*, 2010) propose the execution and testing of parts of the transformation in isolation in order to reduce complexity. However, no systematic method is provided on how to choose these parts of the transformation, which will be tested. In this work we propose the use of traceability information as a possible way to identify problematic parts of a transformation and then use an appropriate transformation testing approach to test extensively these parts.

When a transformation is executed an internal trace is produced by the transformation engine. To identify problematic parts of a transformation, a case-specific traceability metamodel can be possibly used. This is achieved by checking whether the internal trace of the transformation execution conforms to the traceability metamodel. This approach is based on two main assumptions. First, a detailed traceability metamodel needs to be specified before the testing procedure. This traceability metamodel captures all the valid relationships that can exist between two models which conform to the metamodels of interest. Second, we assume that the traceability metamodel is correct and captures all the valid relationships between the two metamodels of interest.

A traceability metamodel includes trace links, which capture valid traceability relationships between two models. In the proposed approach, if a transformation rule generates a traceability link, which can not be matched to any of the existing trace link types in the traceability metamodel, then this rule needs to be tested further. To match the internal trace generated by a transformation engine to the trace links in the traceability model, we follow the same method used in section 4.2, where we matched the internal trace generated by a transformation engine to trace link types in a TML model in order to generate a traceability model. When a transformation is executed an internal trace is generated by the transformation engine and then the transformation postprocessor creates a *GTrace* model, which is a generic traceability representation irrespective of the language used to express the transformation. Thereafter, the postprocessor attempts to find candidate link types in the TML model for the various trace links in the *GTrace* model. This is achieved by comparing the link end types and cardinalities of the various *GTraceLink* with the the link end types and cardinalities of the link types in the TML model. If a match is found, then it is assumed that the *GTraceLink* corresponds to that link type. Otherwise, the *GTraceLink* is flagged as *invalid*. These *invalid GTraceLinks* are indications of faulty transformation rules.

The traceability model generated after a transformation can indicate faults in the transformation specification in a different way as well. Every TML model is accompa-

nied by two sets of validity constraints as discussed in section 4.1.4.2, namely generic constraints and custom constraints. These constraints check the correctness of the generated traceability model and they extend beyond simple type conformance, which is ensured by the existence of a case-specific traceability model. If the validation of a traceability model fails, this is an indication of a problematic transformation. This is because the execution of the transformation has generated an invalid traceability model. Therefore, the transformation specification must be reviewed and modified accordingly. For debugging purposes, appropriate markers and errors/warning messages can be generated, in order to help the tester identify the faulty section of a transformation specification.

The proposed approach on transformation testing using traceability information can be elucidated by the use of a simple example. In this case as well we will use the working example of this thesis in order to illustrate how problematic sections of transformation specifications can be identified by using traceability information. Consider the scenario where the class model, which is illustrated in figure 4.13 and models the booking system of a restaurant, needs to be transformed to its corresponding component model. The TML model, which defines the valid traceability links between the *ClassMetamodel* and the *ComponentMetamodel* is defined in section 4.1.4.4. The transformation between the aforementioned metamodels expressed in ETL is illustrated in listing 4.19. However, for this example we will introduce on purpose different types of errors in the transformation specification and then we will show how these errors can be identified using the proposed approach.

A possible transformation error might be when the transformation engineer transforms an entity of the source metamodel to an invalid entity in the target metamodel. For this example, consider the scenario where the engineer transforms erroneously instances of the *Class* meta-class to instances of the *Component* meta-class. The transformation rule for this transformation is illustrated in listing 4.23.

Listing 4.23: Faulty Class-2-Component ETL transformation - 1st case

```
1 rule Class2Component
2   transform s : ClassModel!Class
3   to t : ComponentModel!Component {
4     t.name = s.name;
5   }
```

When this transformation is executed the transformation postprocessor attempts to generate a traceability model which conforms to the metamodel specified for this scenario. However, in doing so it can not find any valid link types for the *Class2Component* rule, since there is no link type, whose link ends point to the two meta-classes of the transformation rule. As a result, the transformation postprocessor generates a warning marker next to the rule that caused this problem. This is illustrated in figure 4.20. Hav-

ing this information, the transformation engineer can modify the transformation rule accordingly.

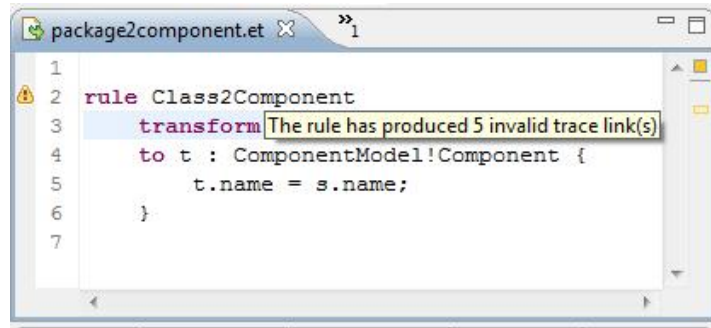


Figure 4.20: Faulty Class-2-Component ETL transformation - 1st case

A different scenario occurs when the transformation engineer transforms an instance of the *Package* meta-class to two instances of the *Component* meta-class. Consider for example the transformation rule illustrated in listing 4.24. This example is very simplistic, but its sole purpose is to illustrate this transformation scenario using the existing example. This transformation will generate a warning, since in the TML model the relationship between instances of the *Package* meta-class and instances of the *Component* meta-class is a 1-to-1 relationship and not 1-to-2 as indicated by this rule. Provided the warning after executing the transformation, the engineer can review and modify the relevant transformation rule accordingly.

Listing 4.24: Faulty Class-2-Component ETL transformation - 2nd case

```
1 rule Class2Component
2   transform s : ClassModel!Package
3   to t1 : ComponentModel!Component, t2 : ComponentModel!Component {
4     t1.name = s.name + '1';
5     t2.name = s.name + '2';
6   }
```

A third scenario is when the transformation generates a traceability model, which conforms to the traceability metamodel, but it does not validate using the correctness constraints, which accompany the traceability model. An example of such a constraint is the constraint illustrated in listing 4.25, which is generated since the *forAll* attribute of the *Package* link end, which belongs to the *Package2Component* link of the TML model, is set to *true*.

Listing 4.25: Constraint generated when the *forAll* attribute is set to *true*

4.4 Traceability usage in the context of MDE

```
1 context Package {
2   constraint OneForEachPackage{
3     check : Package2ComponentTraceLink.all.exists(e|e.Package.target =
         self)
4     message : 'No links of type Package2Component found for Package '
         + self
5   }
6 }
```

A possible transformation rule might transform instances of the *Package* meta-class to instances of the *Component* metaclass, but only when they contain instances of the *Class* meta-class. This constraint is expressed as a guard in line 4 of the transformation illustrated in listing 4.26.

Listing 4.26: Faulty Class-2-Component ETL transformation - 3rd case

```
1 rule Class2Component
2   transform s : ClassModel!Package
3   to t : ComponentModel!Component {
4     guard : s.contents.select(c|c.isTypeOf(Class)).size()>0
5     t.name = s.name;
6   }
```

When this transformation rule is executed, it generates a traceability model which conforms to the traceability metamodel. However, if an instance of the *Package* meta-class does not contain at least one class, then this instance will not be used to generate a corresponding instance of the *Component* meta-class and therefore no trace link will be generated for this particular instance. This does not satisfy the constraint above, which requires that for every instance of the *Package* meta-class there must exist one trace link. Therefore, a validation error will be generated when the validation is executed. Assuming that the TML model captures valid relationships and their constraints between models which conform to the metamodels of interest, this transformation rule must be reviewed and modified accordingly.

4.4.2 Change Propagation with TML

The importance of model transformations in MDE is well documented (e.g. (Mens *et al.*, 2005; Sendall & Kozaczynski, 2003)). There are two types of transformations, namely mapping and update transformations (Mens *et al.*, 2005). Mapping transformation typically transform a source model to a new target model. On the other hand an

update transformation performs in-place modifications of a model. Following (Kolovos *et al.*, 2007), there are two categories of update transformations, namely transformations in the small and transformations in the large. Update transformations in the large apply to a sets of model elements, which is calculated by using a set of well-defined rules. Conversely, update transformations in the small are applied to specific model element, which have been explicitly selected by the user (Kolovos *et al.*, 2007). To summarize the discussion on the different types of model transformation, one can say that mapping transformations take a source model and generate a new target model, update transformations in the large modify a large portion of the model of interest, while update transformation in the small modify only few selected elements of a model.

As developers modify or refactor development entities such as models in order to improve in a disciplined way some of their qualitative attributes, they must ensure to update other system models in order to be consistent with these changes. Many researchers have identified the dangers of not propagating changes (e.g. (Brooks, 1975; Parnas, 1994)). The main consequence of not propagating changes is the introduction of inconsistencies between entities. The use of traceability for identifying impacted artefacts when a change occurs is proposed by many researchers such as (Cleland-Huang *et al.*, 2003) and (Helming *et al.*, 2009). In the context of MDE and model transformations, no inconsistencies can be introduced by mapping transformations, since a new model is generated from scratch when the transformation is executed. In the case of an update transformation in the large inconsistencies change propagation is required since inconsistencies can be introduced by executing such a transformation. Dedicated transformation languages which maintain consistency between two models are proposed in the literature. An example of such a language is the propagating model transformation language (PMT) (Tratt, 2008), in which a model transformation language is described, which is suitable for making suitable updates to models after an initial transformation. To our knowledge, there is no solution proposed to change propagation for update transformations in the small. Therefore, in this work we propose the combination of a task specific language designed for specifying model transformations in the small with TML in order to propagate model changes.

Update transformations are actions that automatically create, update or delete model elements in a user-driven fashion (Kolovos *et al.*, 2007). These actions are commonly referred to as wizards. Since it is quite common in software development that the various models and software artefacts of the development lifecycle are related, when a model is modified or deleted using a wizard, other related artefacts might be affected by this change. Traceability information can be used to identify those related artefacts and modify them accordingly in order to maintain consistency. To demonstrate how this can be achieved we will use the working example of this thesis. Consider the *Class* and *Component* models which describe the restaurant booking system and which are defined in section 4.2.5. These two models conform to the *ClassMetamodel* and

the *ComponentMetamodel*. One simple wizard for the *ClassMetamodel* can be the *RenamePackage* wizard, which applies to a *Package* instance and changes its name. This wizard expressed in the Epsilon Wizard Language (EWL) (Kolovos *et al.*, 2007) is illustrated in listing 4.27. This wizard consists of three parts. The *guard* part defines the types of the elements the wizard applies to. The *title* part provides a short, human-readable description of the wizard's functionality. Finally, in the *do* part of the wizard, the set of actions to be performed to the selected model element(s) is specified.

Listing 4.27: *RenamePackage* wizard expressed in EWL

```
1 wizard RenamePackage {
2   guard : self.isKindOf(Package)
3   title : 'Rename Package'
4   do {
5     var newName = UserInput.prompt("New name", "Select a new name");
6     self.name = newName;
7   }
8 }
```

If a developer uses this wizard to rename a *Package* instance in a class model, then an inconsistency can be introduced. This can happen in the case where we require that the name of a *Package* and the name of its corresponding *Component* should be the same. A solution could have been to regenerate the component model once the class model has been edited. Such a solution is not always ideal, because by reexecuting the transformation in order to propagate the change will cause any manual modification of the component model to be lost. In addition, if the two example models were substantially larger, then reexecuting the entire transformation for such small changes would not be the most efficient approach in terms of computation resources.

In this work we propose the combination of such wizards with TML models in order to maintain consistency in such scenarios. Using the case-specific traceability information specified in a TML model, the wizard can be written in such a way, that in addition to rename the selected *Package* instance, it renames all the *Component* instances which are linked to it using a *Package2ComponentTraceLink*. This is illustrated in listing 4.28, where lines 7-9 select all the instances of *Component*, which are linked to the selected *Package* instance, and rename them accordingly.

Listing 4.28: *RenamePackage* wizard in combination with TML

```
1 wizard RenamePackage {
2   guard : self.isKindOf(Package)
3   title : 'Rename package'
4   do {
```

```
5   var newName = UserInput.prompt("New name", "Select a new name");
6   self.name = newName;
7   for (p2c in Trace!Package2ComponentTraceLink.all.select (p2c|p2c.
      EClass.target = self)) {
8     p2c.Component.target.name = newName;
9   }
10  }
11 }
```

4.5 Chapter Summary

In this chapter, we have presented the main contributions of this thesis. In section 4.1, we have defined and presented TML, which is a metamodeling language for defining case-specific traceability metamodels. TML can be used to generate traceability metamodels and their accompanying correctness constraints. In section 4.2, we have presented how TML models can be used to recover traceability information between two models in a semi-automatic manner. The proposed approach supports both the Dependency and Generative viewpoints and it can be extended to support multiple model management tasks. In section 4.3 of this chapter, a state-based approach to traceability maintenance has been presented based on TML. Finally, in section 4.4 we have proposed two possible uses of traceability information in the context of MDE. The first one has to do with white-box testing and how traceability information can be used to guide the tester to identify problematic sections of a transformation specification. The second one is related to the use of traceability information in update transformations in the small in order to propagate changes to other models.

Chapter 5

Reference Implementation

In chapter 4, TML and its use to support the main traceability activities have been discussed in detail. To build confidence on the feasibility and applicability of the proposed approach a reference implementation of the infrastructure has been constructed.

For this implementation, a combination of Java (Oracle, 2011), the Epsilon framework (Kolovos & Paige, 2010) and the EMF (Eclipse Foundation, 2010a) has been used. Java has been selected as the main language of choice for the reference implementation for several reasons. First, Java is a robust object-oriented language for which there are high-quality open source development tools. One of the most popular tools and the one, which was used for this work, is the Eclipse JDT Eclipse Foundation (2011c). Moreover Java is used by the most widely modelling frameworks such as EMF. The API provided by EMF, has been used extensively for developing the reference implementation, and it is written in Java.

In addition to Java, the Epsilon framework has been selected for developing the various model management tasks, such as model transformations and comparisons. In particular cases, Epsilon has been extended with TML-specific functionality. Such an example is the run configurations developed for ETL and ECL in order to support automatic recovery of trace links. In principle any other model management framework could have been used instead of Epsilon. However, Epsilon was preferred for two main reasons. The first reason had to do with the familiarity of the authors with Epsilon. Second, Epsilon offers some unique features; while those are not essential for implementing the proposed approach they are desirable and useful. One such feature is the support in Epsilon for interactive model management. In a number of occasions, during the execution of a model management task, there is a need to interact with the user to resolve issues that the program cannot decide automatically. Epsilon provides user input for all of its model management tasks. This functionality can be very helpful in the case of traceability support, since due to the informal and complicated nature of traceability activities, it is often the case that user input is required. Since this is a built-in function

in Epsilon, no extra implementation effort was required for adding using interaction features. Another useful aspect of Epsilon is its focus on reusability and modularity. This is due to its architecture, which will be presented in the next section. This feature was useful for developing traceability support, because in Epsilon complex model queries are abstracted in succinct model operations.

Finally, EMF was used as the modeling framework for this work. Similarly to choosing Epsilon as the model management framework, EMF was chosen because it is familiar to the author and because it is supported by open-source high-quality tools and editors. In principle, any other modeling framework could have been used for developing the reference implementation for the proposed approach.

This chapter presents an overview of the reference implementation and highlights noteworthy aspects of it. The rest of the chapter is organized as follows.

5.1 Eclipse Platform

The reference implementation is built atop the Eclipse platform (Eclipse Foundation, 2011a). Eclipse is an open-source software development environment, which consists by an integrated development environment (IDE) and an extensible plug-in system. The Eclipse plug-in system allows developers to extend the *core* of the tool by contributing functionality in the form of plug-ins. Due to this extensibility mechanisms, a wide-range of tools have been developed atop Eclipse, such as editors and execution engines for programming languages, as well as modelling tools such as EMF and GMF. This modular architecture of Eclipse is illustrated in figure 5.1.

Eclipse is the platform of choice for MDE tools as the majority of contemporary MDE languages and frameworks provide Eclipse-based development tools Kolovos (2008b). Moreover, due to the aforementioned extensibility mechanisms, the development of new tools and their integration with existing tools can be done with little effort. Given these reasons, Eclipse was chosen as the platform atop which the reference implementation is developed.

5.2 Eclipse Modeling Framework

The EMF project is a modeling framework and code generation facility for building tools and other applications based on a structured data model. Typically, a model specification is described in the XML Metadata Interchange (XMI) (Group, 2011). EMF provides tools and support to produce a set of Java classes for the model. Moreover, it provides a set of adapter classes that enable the viewing and command-based editing of the model and its editor.

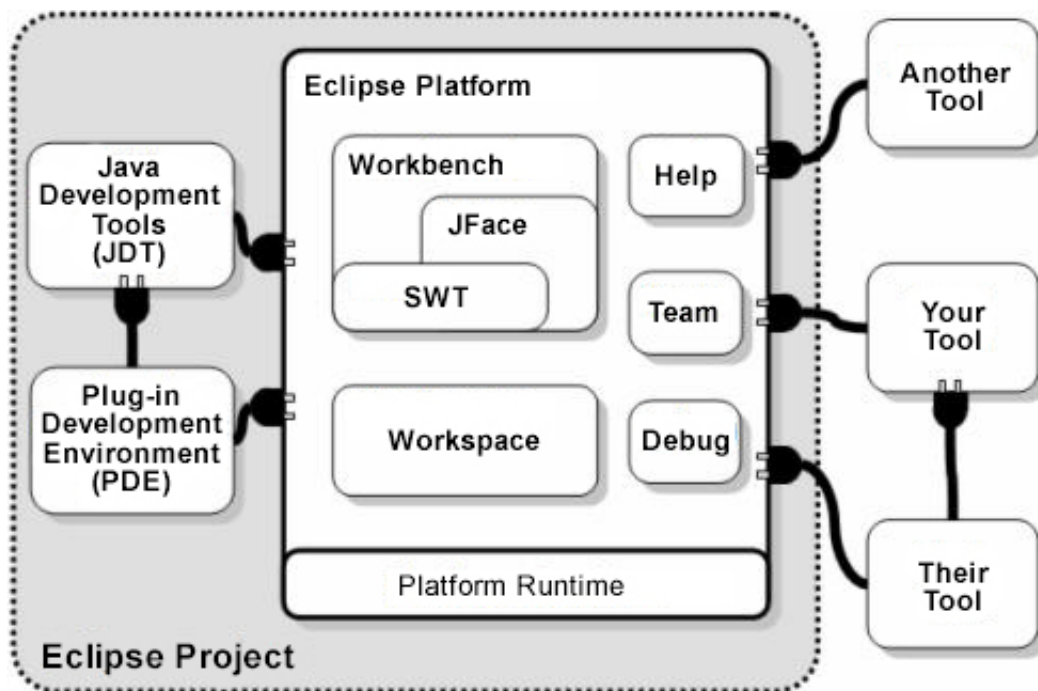


Figure 5.1: The architecture of the Eclipse Platform

The metamodel used to define TML is the model used to represent models in EMF, namely Ecore. Moreover, Ecore traceability metamodels are generated from TML models. Ecore is itself an EMF model, and therefore is its own metamodel. Since EMF is the modeling framework support by the Eclipse platform, and we have chosen Eclipse as the basis for our implementation, EMF was selected as the modeling framework for this work.

5.3 Epsilon Framework

Since in this work we are dealing with models, which need to be navigated and manipulated, an appropriate model management framework had to be selected. The model management framework of choice is the Epsilon framework. Epsilon is a family of consistent and interoperable task-specific programming languages which can be used to interact with EMF models to perform common MDE tasks such as code generation, model-to-model transformation, model validation, comparison, migration, merging and refactoring. Epsilon stands for *Extensible Platform of Integrated Languages for mOdel maNagement*.

Following (Kolovos *et al.*, 2007), each model management task is best supported by

a task-specific language. The purpose of Epsilon is to consolidate the common features of the various task-specific languages in one base language and then develop the various model management languages atop it. The base language is EOL. Moreover, Epsilon provides the Epsilon Connectivity Layer (ECL), which abstracts over the different modelling frameworks and enables the Epsilon task-specific languages to uniformly manage models of those frameworks.

Apart from the existing model management languages, Epsilon provides appropriate extensibility mechanisms to implement new task specific languages with minimal replication. The architecture of the Epsilon framework is illustrated in figure 5.2.

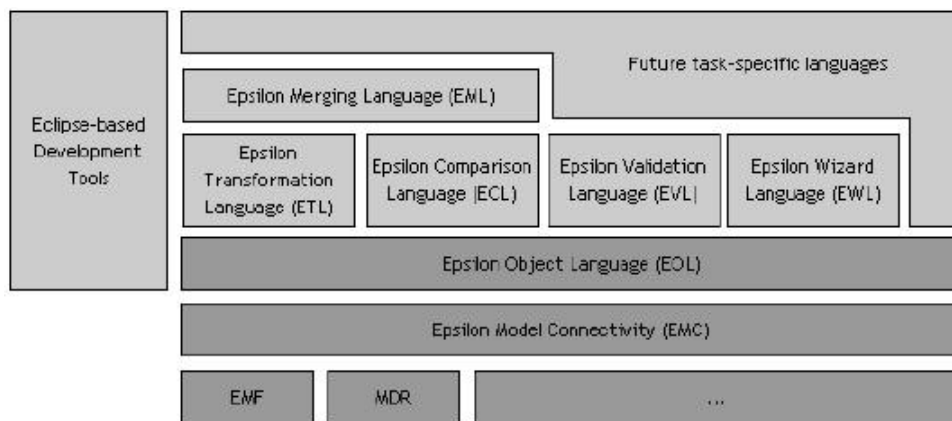


Figure 5.2: The architecture of the Epsilon framework (Kolovos *et al.*, 2007)

At the bottom is the EMC, which abstracts the various modelling technologies such as EMF and Metadata Repository (MDR) (Netbeans, 2003). Atop EMC, EOL is developed, which is the basis for the current and future task-specific languages. Since Epsilon framework is build in Eclipse, it is well integrated with other Eclipse development tools, through the use of the plug-in system provided by the Eclipse platform. The task specific languages used for this work are briefly described in the following.

5.3.1 Epsilon Object Language

EOL (Kolovos *et al.*, 2006a) has been used extensively to implement common model querying and navigation activities. EOL reuses the navigational and querying features of the Object Constraint Language (OCL) (Object Management Group, 2003) and it adds support for statement sequencing and model modification capabilities. Moreover, EOL provides simultaneous access to multiple models which conform possibly to heterogeneous metamodels. Finally, EOL supports reuse by allowing user to define operations, which apply to elements of a specific meta-class. Multiple user-defined operations have

been defined in the context of this work. An example of one of the user-defined operations used in this work is the *createEType()* operation, which is shown in listing D.1.

5.3.2 Epsilon Transformation Language

ETL (Kolovos *et al.*, 2008) is a hybrid, rule-based, model-to-model transformation language built on top of EOL. It is capable of transforming an arbitrary number of source models into an arbitrary number of target models. Moreover, as ETL is based on EOL, it reuses its imperative features to enable users to specify particularly complex, and even interactive, transformations. In the context of this work, ETL is used to define the transformation specification from a TML model to a case-specific traceability metamodel expressed in Ecore. This transformation is presented in section 4.1.4.2.

5.3.3 Epsilon Generation Language

EGL (Rose *et al.*, 2008) is a template-based, model-to-text transformation language for generating code, documentation and other textual artefacts from models. EGL offers model-to-text specific features such as protected regions for mixing generated with hand-written code and template coordination. Similarly to ETL, EGL is built atop EOL, therefore it has access to general model management support. In the context of this work, EGL is used to generate the generic and custom correctness constraints of the traceability metamodel. The transformations for deriving these constraints are presented in section 4.1.4.2.

5.3.4 Epsilon Validation Language

EVL (Kolovos *et al.*, 2007) is the validation language of the Epsilon platform. It is an OCL-like validation language, which in addition supports dependencies between constraints, customizable error messages and specification of *fixes*, which are invoked to repair inconsistencies. Similarly to the other languages of the Epsilon framework, EVL builds on top of EOL. This enables it to evaluate inter-model constraints. In the proposed approach, EVL is used to express correctness constraints which apply to a traceability metamodel. The generation of such constraints is presented in section 4.1.4.2.

5.3.5 Epsilon Comparison Language

ECL (Kolovos, 2009) aims to enable users to specify comparison algorithms in a rule-based manner to identify pairs of matching elements between two models of potentially different metamodels. ECL is a hybrid language with support for the specification of fuzzy matching algorithms as well as with support for interactive matching. ECL is used

in this work to provide traceability support for the Dependency viewpoint. To recover trace links between two models, which co-exist, correspondences between them must be found. ECL is used to express the comparison algorithms between those models and the internal trace, which is returned from the comparison is used to recover the trace links between the models of interest. This approach is presented in section 4.2.3.1.

5.3.6 Epsilon Wizard Language

The final language of the Epsilon framework, which is used in this work, is EWL (Kolovos *et al.*, 2007). EWL is language, whose aim is to support interactive in-place transformations on user-selected model elements. The niche of EWL is the automation of recurring model editing tasks such as model refactorings. EWL is used in this thesis in the context of traceability usage. We propose the use of EWL in combination with traceability models, in order to refactor models and then propagate the changes to affected model elements in other models. The presentation of this approach takes place in section 4.4.2.

5.4 Eclipse Views & Editors

To enable users to define TML models and then manipulate them accordingly, we have reused the basic EMF tree editor. However, the editor has been customised by using the EXtended Emf EDitor (Exeed) (Kolovos, 2007). This is achieved by adding Exeed-specific EAnnotations to the EClasses of TML. These annotations provide instructions about how to format labels and icons of the editor and then Exeed uses this information to visualise the TML models accordingly. A screenshot of the Exeed editor is illustrated in figure 5.3.

Moreover, we have extended Eclipse context menus with TML specific options. In the context of models with the *tml* extension, an option for generating the Ecore meta-model and the accompanying constraints. In the context of models with the *tmltrace* extension, an option for executing the “traceability maintenance” operation. An illustration of this context specific menus is shown in figure 5.4.

Finally, to enable users to locate errors in ETL transformations, we have enhanced the ETL editor with in-place markers as well as with error messages which are printed in the Epsilon console.

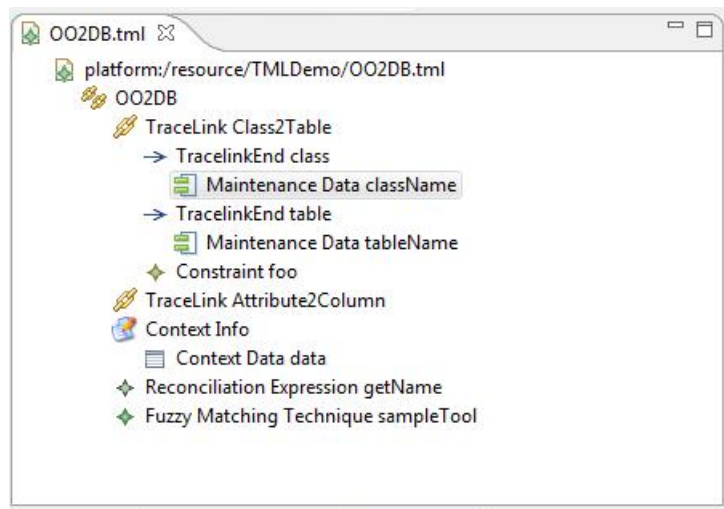


Figure 5.3: Example of the Exeed TML editor

5.5 Launch Configuration Interface

To support the automatic generation of case-specific traceability models, the ETL and ECL launch configuration instances had to be modified. In addition to specifying the models on which the transformation or comparison operates on, the traceability meta-model to which the traceability model will conform, has to be specified. To this end, additional tabs and dialogs has been added to the existing ETL and ECL launch configurations. Figure 5.5 illustrates the modified launch configuration interface of ETL.

5.6 Availability

Initially, the source code and documentation of the implementation was maintained locally. Since November 2010 the Eclipse-based traceability tools have been hosted by “Google Project Hosting” website. The source code for this work can be found at the following address:

<http://tml.googlecode.com/svn/trunk/>

5.7 Chapter Summary

In this chapter, an overview of the reference implementation that enables users to use the proposed approach is provided. Through this chapter, noteworthy implementation

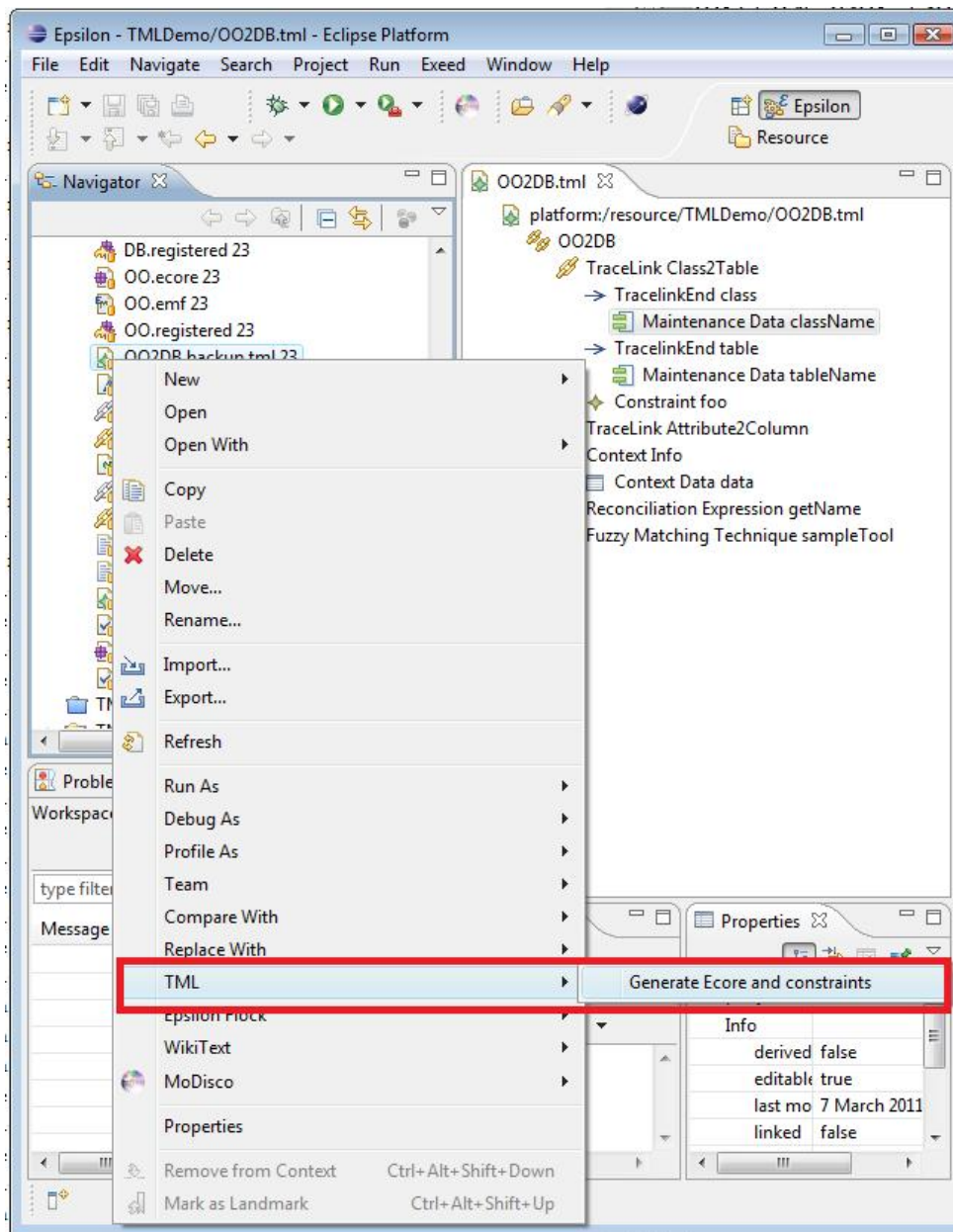


Figure 5.4: Illustration of TML context menu

decisions such as the choice of specific development environments or of specific programming and developing languages has been discussed. The next chapter evaluates the validity of the hypothesis using a number of criteria, including a complex case study which has been realized using the reference implementation.

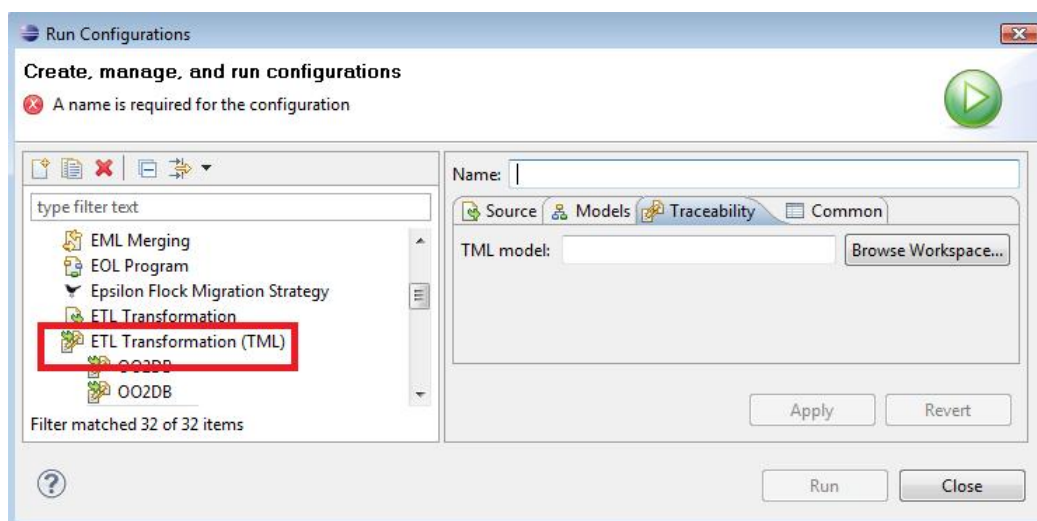


Figure 5.5: Illustration of the modified launch configuration of ETL

Chapter 6

Evaluation

In this thesis a novel approach to model-to-model traceability for MDE processes has been proposed. This chapter describes the means by which the proposed approach was evaluated against the thesis proposition throughout the duration of this research. The thesis proposition was stated as follows:

This thesis demonstrates that a domain specific model-based traceability approach can support and automate the process of rigorously managing the different types of heterogeneous traceability relationships between both derived and initial models in an MDE process.

As discussed in chapter 3, the proposition implies the following challenges:

1. **Model-based Approach:** In the spirit of MDE the proposed approach should be applicable to model artefacts.
2. **Manage Traceability:** The proposed approach should provide support for all four traceability activities, i.e. representation, identification, maintenance and usage of traceability.
3. **Heterogeneous Traceability Relationships:** Using the proposed approach, an engineer should be able to manage traceability relationships between models expressed in heterogeneous languages.
4. **Rigorous:** The proposed approach should support the capture of case-specific or scenario-specific semantics.
5. **Automation:** The proposed approach should provide semi-automatic identification and maintenance of traceability information

6. Derived and Initial Models: The proposed approach should be suitable for traceability between models which are generated automatically by applying model operations on other models (derived models), as well as between models, which are created manually by engineers (initial models).

The aforementioned challenges are addressed by four distinct strands within the thesis, namely:

- The representation of traceability using a dedicated metamodeling language, namely TML, which focuses on specifying case-specific traceability metamodels. This strand is described in section 4.1.
- Traceability identification using traceability models with rich semantics described in section 4.2.
- Traceability maintenance by reflectively evolving the traceability model described in section 4.3.
- Traceability usage is presented in section 4.4.

The evaluation of the proposition was focused on examining the application of the framework with respect to the following two concerns. First the feasibility of the approach and second the benefits yielded by applying the approach in order to address the challenges inherent in the proposition

6.1 Means of Evaluation

Various means of evaluation were employed during the different stages of the research. These included the following:

- Use of traceability scenarios
- Peer review
- Case study

In the following sections, we will explain how we evaluated the proposed work using the above means, as well as the results of this evaluation.

6.1.1 Traceability scenarios

The initial means of evaluation for the various concepts of this thesis were examples and traceability scenarios. New concepts were tested by their application in such scenarios. If proof of the applicability of a concept was provided, then the concept was accepted. On the other hand, if a counter-example was found, these concepts were rejected as inappropriate and inefficient.

Examples of the traceability scenarios used include:

- Traceability between a Class schema and a relational database model.
- Traceability between Goal models and Class schemas.
- Traceability between a Table model and an HTML Table specification.

Examples were only an initial evaluation technique. The application of the various concepts of this thesis was also evaluated in a larger scale case study, which is presented in section 6.1.3.

6.1.2 Peer review

The proposal and the design of the proposed approach was inherently a difficult subject to evaluate due to the fact that it is infeasible to obtain a statistically significant sample of case studies. For this reason, peer review was considered to be an important means of evaluation of the validity of the approach. The results of this research have been presented in a number of academic papers in journals, international conferences and workshops. The feedback related to this work was positive and it had to do with the feasibility of the approach as well as with the potential benefit gained from its application. The author's publications, which are directly related to this thesis, are listed below.

- Nicholas Drivalos, Richard F. Paige, Kiran Fernandes and Dimitrios S. Kolovos. *Towards Rigorously Defined Model-to-Model Traceability*, in Proc. 4th Workshop on Traceability, ECMDA'08, Berlin, Germany, June 2008
In this paper the main motivation for case-specific traceability support was provided.
- Nicholas Drivalos, Dimitrios S. Kolovos, Richard F. Paige, Kiran J. Fernandes. *Engineering a DSL for Software Traceability*, in Proc. 1st International Conference on Software Languages Engineering, SLE '08, Toulouse, France, Sept 2008
In this paper the abstract syntax and the semantics of TML are presented.

- Steffen Zschaler, Dimitrios S. Kolovos, Nikolaos Drivalos, Richard F. Paige and Awais Rashid. *Domain-Specific Metamodelling Languages for Software Language Engineering*, in Proc. 2nd International Conference on Software Language Engineering, Colorado, USA, October 2009

In this paper, the author contributed a discussion on TML as a special case of a *Domain Specific Metamodelling Language* (DSM2L). The term DSM2L was coined first in (Drivalos *et al.*, 2009).

- Richard F. Paige, Nicholas Drivalos, Dimitrios S. Kolovos, Chris Power, Goran K. Olsen and Steffen Zschaler. *Rigorous Identification and Encoding of Trace-Links in Model-Driven Engineering*, Journal of Software and Systems Modelling, Springer, 2010 - accepted and to appear - DOI: 10.1007/s10270-010-0158-8

The contribution of the author in this paper is a discussion on what constitutes traceability information with rich semantics.

- Louis M. Rose, Dimitrios S. Kolovos, Nicholas Drivalos, James R. Williams, Richard F. Paige, Fiona A.C. Polack, and Kiran J. Fernandes. *Concordance: An Efficient Framework for Managing Model Integrity*, in Proc. 6th European Conference on Modelling Foundations and Applications (ECMFA), June 2010, Paris, France

In this work, the author has contributed to the discussion about the requirements of a framework for managing model integrity, as well as how such a framework can be used to maintain traceability models.

- Nicholas Drivalos-Matragkas, Dimitrios S. Kolovos, Richard F. Paige, Kiran J. Fernandes. *A state-based approach to traceability maintenance*, in Proc. of the 6th ECMFA Traceability Workshop, Paris, France, June 2010 In this work, the approach for maintaining the integrity of traceability models is presented.

Through a web survey, it was found that the above work has been cited in a number of external publications (e.g. (Dhoolia *et al.*, 2010; Schwarz *et al.*, 2009; Seibel *et al.*, 2010; Williams & Polack, 2010; Winkler & von Pilgrim, 2009). This provides further evidence that this work is well-communicated and established within the MDE research community.

6.1.3 Case study

The third means of evaluation was a complex case study. The complexity of it is related to the size of the produced traceability model as well as to the nature of the traceability relationships.

In this case study, traceability support is provided for the EuGENia (Kolovos *et al.*, 2009) tool using TML. EuGENia is a tool that automatically generates all the models

needed to implement a Graphical Modelling Framework (GMF) (Eclipse Foundation, 2011b) editor from a single annotated Ecore metamodel. In the sequel, we will describe the context of this case study by introducing what GMF and EuGENia are and how EuGENia works. Finally we will present how EuGENia can be enhanced when combined with TML.

6.1.3.1 Graphical Modelling Framework

GMF provides a generative component and runtime infrastructure for developing graphical editors based on the EMF and the Graphical Editing Framework (GEF) (Moore, 2004). The aim of GMF is to reduce the time and effort required to develop diagram-based editors for modelling languages.

In order to implement a graphical editor with GMF, developers need to construct a number of intermediate models that specify various characteristics of the visual editor. These models are then combined and used eventually as input in a set of model-to-text transformations in order to generate the Eclipse plug-ins that contain the Java code, which implements the editor. The process of developing a graphical editor using GMF is illustrated in figure 6.1.

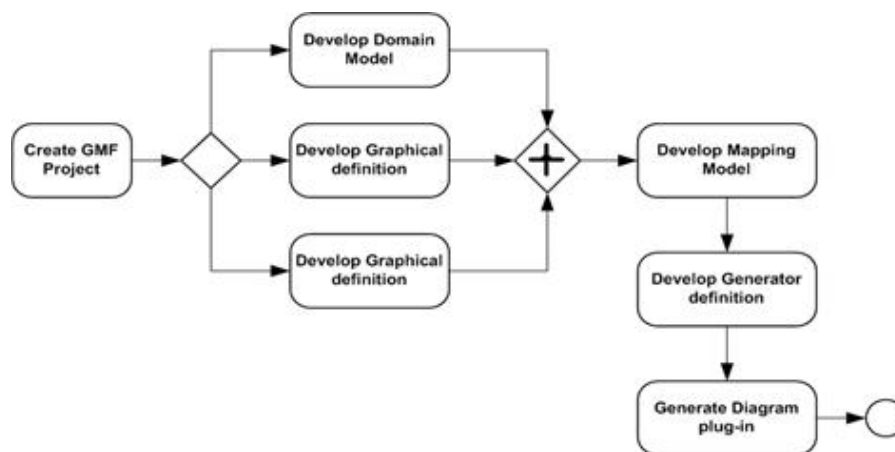


Figure 6.1: GMF Overview (Eclipse Foundation, 2011b)

In this diagram, the main components and models used during GMF-based development are illustrated. The first model, which is required by GMF, is the domain model. This model is expressed in Ecore and its purpose is to capture the main concepts of the modelling language for which the editor is built. Apart from the domain model, the graphical and tooling definitions of the language have to be defined. The first model which captures such information is the *graphical definition model* (gmfgraph). This model consists of the graphical elements, which will be used in the diagram to display

classes of the domain model. Moreover, the tool definition model (*gmftool*) contains optional tooling definitions for the design of the palette and other aspects of the editor such as menus and toolbars. The final model used during GMF-based development is the mapping model (*gmfmap*), whose purpose is to link together the graphical and tooling definitions to the selected domain model. Once the appropriate *gmfmap* model is defined, GMF provides a generator model (*gmfgen*) to allow implementation details to be defined for the generation phase. This generation model is then consumed by a set of model-to-text transformations that eventually generate the Java code and Eclipse plugins, which realise the graphical editor.

To support this procedure, GMF provides a wizard that can facilitate the generation of the aforementioned models from the domain model. The generated models have then to be refined manually using dedicated tree editors. (Kolovos *et al.*, 2009) have identified a number of problems associated to this process. First, the refinement of the models generated by the wizard, requires a good knowledge of the respective metamodels (i.e. the *gmfgraph*, *gmftool* and *gmfmap* metamodels). However, those metamodels are far from trivial. In addition, in case a developer makes a logical error in one of the models, GMF either produces low level error messages, which are not really helpful for a novice user, or it proceeds with the code generation and generates erroneous code. Furthermore, in the case of complex domain models, the GMF wizard fails to yield useful results (Wienands & Golm, 2009). This can be attributed to the fact that very little can be inferred about the graphical syntax of a notation based on the abstract syntax alone. As a consequence, a big part of the GMF models has to be handcrafted. Finally, maintaining the handcrafted GMF-specific models is quite challenging, since GMF does not provide any reconciler that can update these models automatically when the domain model changes. Therefore, once the models are customised after their initial generation by the GMF wizard, they have to be maintained manually.

All of the aforementioned issues make implementing a graphical editor with GMF a laborious and error prone task, particularly so for the inexperienced developer. In order to address the issues related to GMF and enable developers to implement fully-functional GMF editors with minimal effort, (Kolovos *et al.*, 2009) have developed a tool called EuGENia. This tool will be presented in the sequel.

6.1.3.2 EuGENia

Despite its issues, GMF is currently one of the most powerful and widely used open-source graphical editor frameworks (Kolovos *et al.*, 2010). The aim of EuGENia is to shield developers from the complexity of GMF and address the highlighted challenges.

EuGENia provides a higher level of abstraction for the definition of the graphical concrete syntax of modelling languages. This is achieved by using annotations at the metamodel level. These annotations provide a generic and flexible mechanism for extending the metamodel with visual concrete syntax information. Once this informa-

tion is specified, EuGENia can automatically produce the required GMF intermediate models which are necessary in order for GMF to generate a fully-functional visual editor. This generation is made feasible by using automated model-to-model and in-place transformations to generate, in a consistent and repeatable manner, the platform-specific models required by GMF.

Following (Kolovos *et al.*, 2010), to automate the construction of the GMF-specific models, the Ecore domain model is annotated with high level GMF-specific information and then a model-to-model transformation is used to generate the tooling, graph and mapping GMF models. In the next step the gmfmap model is transformed to the generator model (gmfgen) using the GMF built in transformation. Finally, an in-place update transformation is applied to the gmfgen in order to specify some of the graphical syntax configuration options. The EuGENia workflow is illustrated in figure 6.2.

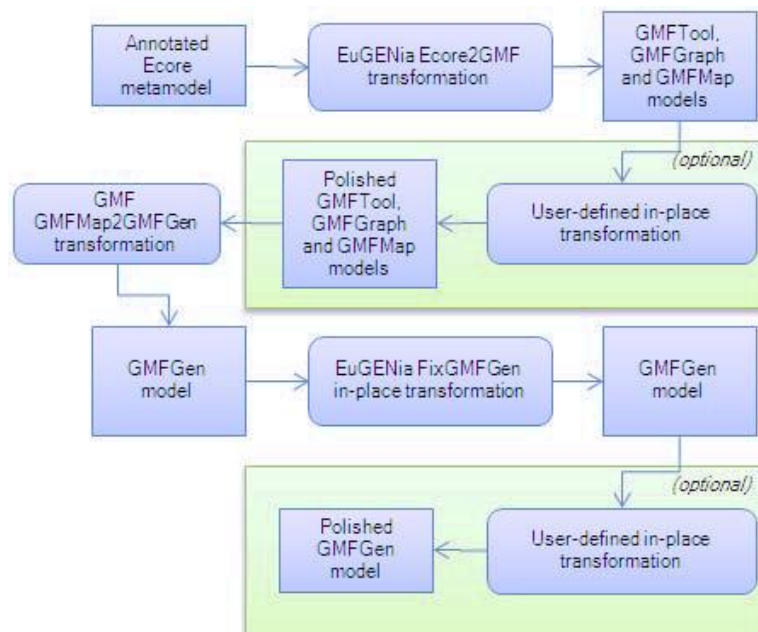


Figure 6.2: EuGENia workflow (Kolovos *et al.*, 2010)

The EuGENia transformations are implemented using the imperative model transformation language. The choice of an imperative language over a rule-based language has to do with the high complexity of the transformation and the need for low-level control of the execution flow.

One limitation of EuGENia is the fact that it does not provide support for all the features of GMF. Instead it is focused on the most commonly used features for implementing a visual editor (Kolovos *et al.*, 2009). For cases where the concrete syntax of the notation of interest can not be fully supported by EuGENia, developers need to mod-

ify the generated from EuGENia models manually. A problem which can arise when changes occur at the domain model, involves keeping the domain and GMF models consistent without affecting any manual modifications of the GMF models. That is, when a model element in the domain model is refactored any related model elements of the GMF models must be modified accordingly in order for the models to remain consistent. There are two options for updating the GMF models. First, the transformations can be re-executed so that the updates are reflected in the new GMF models. However, any manual modifications of the initial intermediate models (for example of some features, which are not supported by EuGENia) will be lost. In the second option, the developer can refactor the domain model and then manually update the generated models. However, this is a tedious and error prone task, since the annotations supported by EuGENia are not a 1-1 mapping with the features of GMF. Therefore, finding and modifying the appropriate elements in the GMF models is not a trivial task.

A possible solution to this issue is the use of traceability information and of dedicated wizards in order to refactor the domain model and update the GMF models automatically, without re-executing the EuGENia transformations. This is the scenario we will use in order to apply and evaluate our approach. Below we will examine how combining our approach with EuGENia can help solve the aforementioned problem. To this end, we will use a concrete example in which we implement an editor for a filesystem metamodel.

6.1.3.3 The filesystem metamodel

In this case study, we will use the modelling scenario described in (Kolovos *et al.*, 2009). In this scenario, EuGENia is used to generate a graphical editor for a filesystem domain model. As implied by its name, this metamodel specifies the concepts involved in a file system and their relationships. In listing 6.1, the filesystem metamodel is presented expressed in the Emfatic (alphaWorks, 2005) textual notation. The root concept of this metamodel is the *Filesystem* class. A filesystem can contain any number of *Drives* and *Syncs*. *Files*, *Shortcuts* and *Folders* are special types of *Drives*. A *Sync* class represents a synchronization relationship between a source and a target files, while a *Shortcut* is a special file that points to a target file.

Listing 6.1: The filesystem metamodel (Kolovos *et al.*, 2009)

```
1 @namespace(uri="filesystem", prefix="filesystem")
2
3 package filesystem;
4
5 class Filesystem {
6     val Drive[*] drives;
7     val Sync[*] syncs;
```

```

8 }
9
10 class Drive extends Folder {
11 }
12
13 class Folder extends File {
14     val File[*] contents;
15 }
16
17 class Shortcut extends File {
18     ref File target;
19 }
20
21 class Sync {
22     ref File source;
23     ref File target;
24 }
25
26 class File {
27     attr String name;
28 }

```

To generate the graphical editor for this domain model with EuGENia, one must first decide on how each concept of the metamodel is mapped to a visual construct in the editor. After taking this decision, the metamodel expressed in Emfatic is annotated accordingly. The annotation `gmf.diagram` is attached to the *Filesystem* class in line 5. This annotation specifies that this class will be represented as a diagram. In line 30, the `gmf.node` is attached to the *File* class and it specifies that each *File* will be represented as a node containing a label, whose value is its name attribute. Since *Drive*, *Folder*, and *Shortcut* extend the *File* class they will be represented as nodes as well, since they inherit the visual syntax of their super-class. The `gmf.link` annotation, which is specified in line 24, specifies that a *Sync* will be represented as a dotted link between a source and a target file. Moreover, the same annotation is defined for *target* property of the *Shortcut* class in line 20 and it specifies that this property is represented as a dashed link with an arrowhead pointing at the target *File*. Finally, the `gmf.compartment` annotation specifies that each *Folder* will contain a compartment that will host the *Files* contained in the objects contents reference.

Listing 6.2: The filesystem metamodel with EuGENia-specific annotations (Kolovos *et al.*, 2009)

```
1 @namespace(uri="filesystem", prefix="filesystem")
```

```
2 @gmf(foo="bar")
3 package filesystem;
4
5 @gmf.diagram(foo="bar")
6 class Filesystem {
7     val Drive[*] drives;
8     val Sync[*] syncs;
9 }
10
11 class Drive extends Folder {
12 }
13
14 class Folder extends File {
15     @gmf.compartment(foo="bar")
16     val File[*] contents;
17 }
18
19 class Shortcut extends File {
20     @gmf.link(target.decoration="arrow", style="dash")
21     ref File target;
22 }
23
24 @gmf.link(source="source", target="target", style="dot", width="2")
25 class Sync {
26     ref File source;
27     ref File target;
28 }
29
30 @gmf.node(label = "name")
31 class File {
32     attr String name;
33 }
```

Once the Ecore model is annotated with the appropriate EuGENia annotations, the intermediate GMF models can be automatically generated and the graphical editor can be derived. This editor is illustrated in figure 6.3.

During this process no traceability information between the filesystem domain model and the intermediate GMF models (gmftool, gmfgraph, gmfmap) is maintained. As a result, if a change occurs in the filesystem model, we either have to regenerate everything or we have to manually modify the affected models. In order to provide automatic support for this kind of refactorings, we will enhance EuGENia with TML so that when

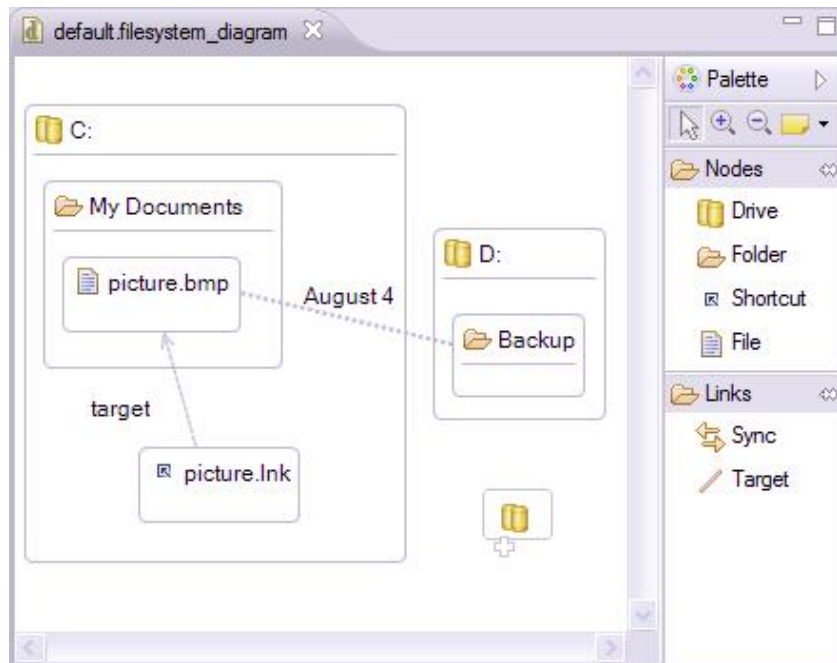


Figure 6.3: Graphical editor of the filesystem DSL (Kolovos *et al.*, 2010)

a model element in the filesystem is modified, the corresponding elements in the intermediate GMF models are modified accordingly.

6.1.3.4 Defining Traceability between Ecore and GMF Models

The first step of the proposed approach is to define the valid traceability relationships between the Ecore domain model and the GMF models as well as any correctness constraints that might apply to this scenario. To do this we shall use the TML tree editor in order to specify the TML case-specific traceability model. This model is illustrated in figure 6.4. At this point it should be noted that for brevity the provided TML model does not capture all existing relationship types between the an Ecore model and the GMF models. By using a subset of the existing relationship types we avoid the introduction of unnecessary complexity in the discussion.

This TML model captures the following trace link types:

- *EPackage2Canvas*: this link type represents links between an EPackage of the domain model and a Canvas of the gmfgraph model. The *forAll*, *Unique* and *ofTypeOnly* attributes are set to true for both of the link ends. The EPackage link end has a maintenance attribute called *Name*, whose value can be retrieved by using the *getName* reconciliation expression. Similarly, the Canvas link end has

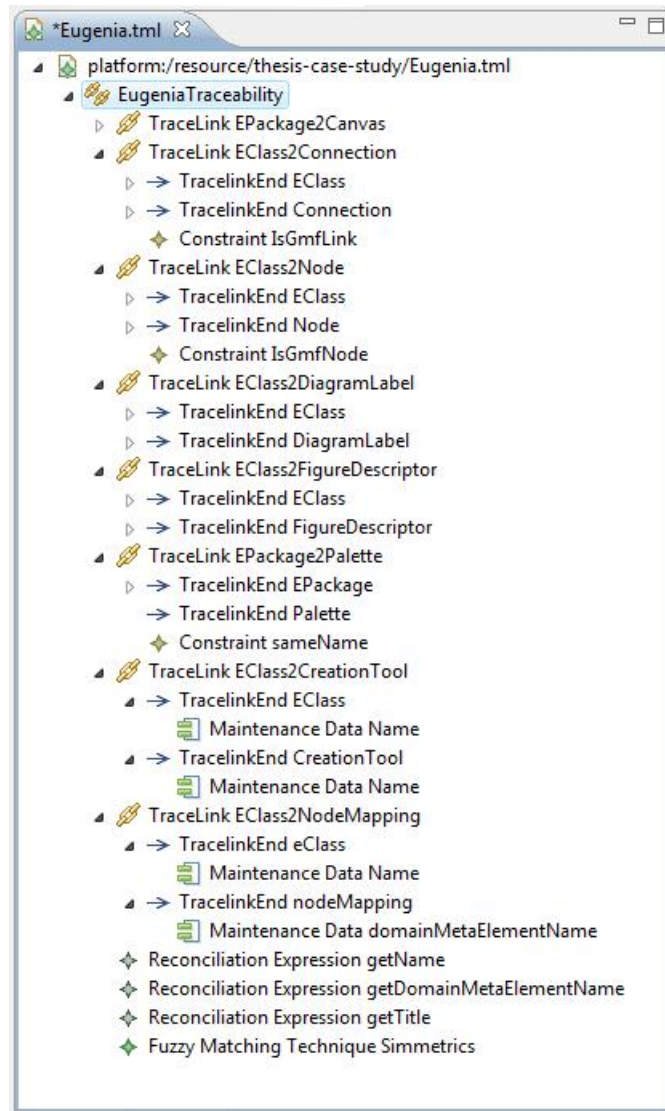


Figure 6.4: TML model for filesystem case study

also the *Name* maintenance data attribute, whose value can be retrieved by the same reconciliation expression. Moreover, the *sameName* constraint is attached to this link. This constraint checks whether the EPackage and the Canvas have the same name.

- *EClass2Connection*: this link type represents links between an EClass of the domain model and a Connection of the gmfmap model. The *forAll* attribute for the EClass link end is set to false, while the same attribute for the Connection class is set to true. Furthermore, the *Unique* and the *ofTypeOnly* attributes are set to true for both of the link ends. The EClass link end has a maintenance attribute called *Name*, whose value can be retrieved by using the *getName* reconciliation expression. Similarly, the Connection link end has also the *Name* maintenance data attribute, whose value can be retrieved by the same reconciliation expression. The *sameName* constraint is attached to this link as well.
- *EClass2Node*: this link type represents links between an EClass of the domain model and a Node of the gmfgraph model. The *forAll* attribute for the EClass link end is set to false, while the same attribute for the Node class is set to true. The *Unique* and the *ofTypeOnly* attributes are set to true for both of the link ends. The EClass link end has a maintenance attribute called *Name*, whose value can be retrieved by using the *getName* reconciliation expression. Similarly, the Node link end has also the *Name* maintenance data attribute, whose value can be retrieved by the same reconciliation expression. Moreover, the *IsGmfNode* constraint is attached to this link. This constraint checks whether the annotation attached to the EClass link end is a *gmf.node* EuGENia annotation.
- *EClass2DiagramLabel*: this link type represents links between an EClass of the domain model and a DiagramLabel class of the gmfgraph model. The *forAll* attribute for the EClass link end is set to false, while the same attribute for the DiagramLabel class is set to true. Furthermore, the *Unique* and the *ofTypeOnly* attributes are set to true for both of the link ends. The EClass link end has a maintenance attribute called *Name*, whose value can be retrieved by using the *getName* reconciliation expression. Similarly, the DiagramLabel link end has also the *Name* maintenance data attribute, whose value can be retrieved by using the same reconciliation expression.
- *EClass2FigureDescriptor*: this link type represents links between an EClass of the domain model and a FigureDescriptor of the gmfgraph model. The *forAll* attribute for the EClass link end is set to false, while the same attribute for the FigureDescriptor class is set to true. The *Unique* and the *ofTypeOnly* attributes are set to true for both of the link ends. The EClass link end has a maintenance

attribute called *Name*, whose value can be retrieved by using the *getName* reconciliation expression. Similarly, the *FigureDescriptor* link end has also the *Name* maintenance data attribute, whose value can be retrieved by the same reconciliation expression.

- *EPackage2Palette*: this link type represents links between an *EPackage* of the domain model and a *Palette* of the *gmftool* model. The *forAll*, *Unique* and *ofTypeOnly* attributes are set to true for both of the link ends. The *EPackage* link end has a maintenance attribute called *Name*, whose value can be retrieved by using the *getName* reconciliation expression. Similarly, the *Palette* link end has also the *Name* maintenance data attribute, whose value can be retrieved by the same reconciliation expression. Finally, the *sameName* constraint is attached to this trace link, in order to check whether the *EPackage* and the *Palette* share the same name.
- *EClass2CreationTool*: this link type represents links between an *EClass* of the domain model and a *CreationTool* in the *gmftool* model. The *forAll* attribute for the *EClass* link end is set to false, while the same attribute for the *CreationTool* class is set to true. Moreover, the *Unique* and the *ofTypeOnly* attributes are set to true for both of the link ends. The *EPackage* link end has a maintenance attribute called *Name*, whose value can be retrieved by using the *getName* reconciliation expression. Similarly, the *CreationTool* link end has also the *Name* maintenance data attribute, whose value can be retrieved by the same reconciliation expression.
- *EClass2NodeMapping*: the final trace link type of this TML model represents links between an *EClass* of the domain model and a *NodeMapping* of the *gmfmap* model. The *forAll* attribute for the *EClass* link end is set to false, while the same attribute for the *NodeMapping* class is set to true. The *Unique* and the *ofTypeOnly* attributes are set to true for both of the link ends. The *EClass* link end has a maintenance attribute called *Name*, whose value can be retrieved by using the *getName* reconciliation expression. The *NodeMapping* link end has the *domainMetaElementName* maintenance data attribute, whose value can be retrieved by the *getDomainMetaElementName* reconciliation expression (`self.domainMetaElement.name`). This expression retrieves the name of the element of the domain model to which *NodeMapping* link end corresponds.

Once the TML model is defined, an *Ecore* traceability metamodel and its accompanying correctness constraints can be generated by using the built-in TML transformations. The generated traceability metamodel captures the valid trace links between an *Ecore* model and the GMF models (i.e. *gmfgraph*, *gmftool*, *gmfmap*). An illustration of this metamodel is provided in figure 6.5.

There are two sets of constraints generated from the above TML model. The first set consists of those that are automatically generated from the various attributes of the

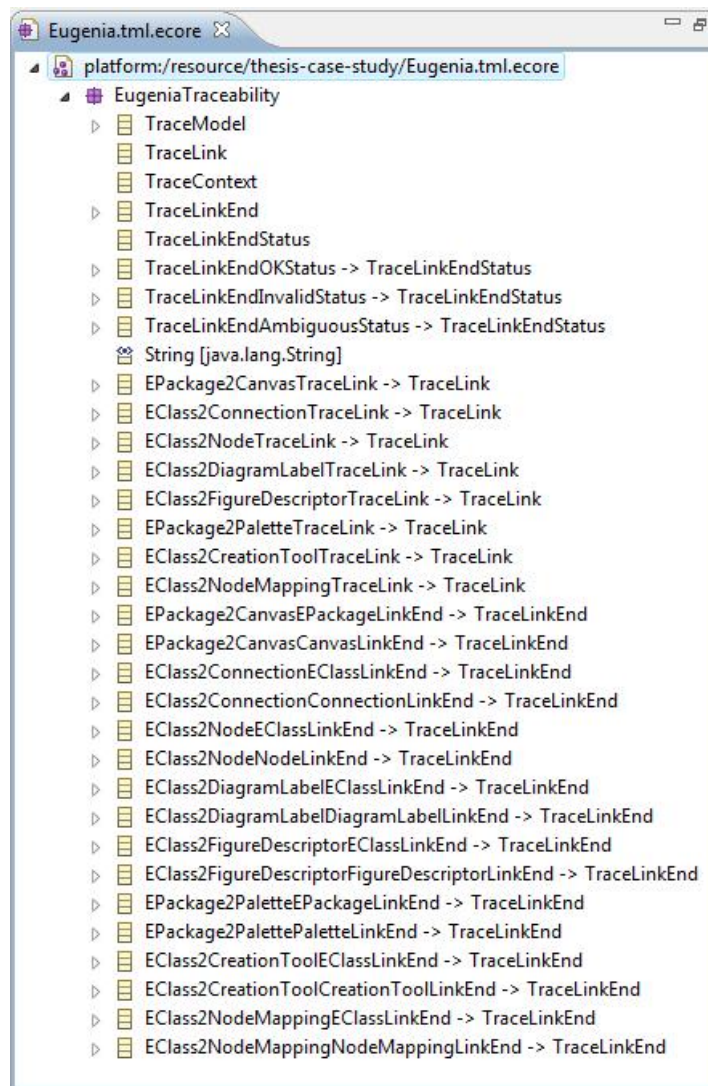


Figure 6.5: Illustration of the EuGENia Ecore metamodel

TML model. The second set consists of constraints that can not be fully generated from the TML model. In this case a skeleton of the constraints is generated and then the body of those constraints has to be written by the developer. An example of a constraint belonging to the first set is illustrated in listing 6.3. This constraint captures the fact that a Node must be related to only one EClass via a EClass2Node trace link.

Listing 6.3: The *UniqueNode* constraint

```

1 context Node {
2   constraint UniqueNode{
3
4     guard : self.isTypeOf(Node)
5
6     check : EClass2NodeTraceLink.all.select(e|e.Node.target = self).
          size() < 2
7
8     message : 'Multiple links of type EClass2Node found for Node ' +
          self
9   }
10 }

```

Another example of generated constraint is the *OneForEachDiagramLabel* constraint illustrated in listing 6.4. This constraint checks, whether every instance of the *DiagramLabel* metaclass in a model is related to an instance of the *EClass* metaclass via an instance of the *EClass2DiagramLabel* trace link.

Listing 6.4: The *OneForEachDiagramLabel* constraint

```

1 context DiagramLabel {
2   constraint OneForEachDiagramLabel{
3
4     guard : self.isTypeOf(DiagramLabel)
5
6     check : EClass2DiagramLabelTraceLink.all.exists(e|e.DiagramLabel.
          target = self)
7
8     message : 'No links of type EClass2DiagramLabel found for
          DiagramLabel ' + self
9   }
10 }

```

The second set of constraints is generated using the *Constraint* model elements of the TML model. An example of such a user-defined constraint is the *IsGmfNode* constraint, which is illustrated in listing 6.5. This constraint checks whether the *EClass* which is referred by the *EClass* link end of an *EClass2NodeTraceLink* is annotated with the *@gmf.node* annotation. If this *EClass* is not annotated with this annotation, then it should not be linked to a *Node*.

Listing 6.5: The *IsGmfNode* constraint

```

1 context EClass2NodeTraceLink {
2   //The annotation link end must be a gmf.node
3   constraint IsGmfNode {
4
5     check : self.EClass.isNode()
6
7     message : "EClass " + self.EClass.name + " is not annotated as
           @gmf.node"
8   }
9 }

```

In this constraint, the user-defined operation *isNode* is used. This operation is listed in figure 6.6. This operation is declared as *cached*, meaning that it is only executed once for each distinct EClass and subsequent calls on the same target return the cached result. In lines 4 and 5, the operation checks if an EClass is a link or abstract. If it is neither, the operation checks if the *gmf.node* annotation is applied to the target EClass and if it returns *true*

Listing 6.6: The *isNode* EOL user-defined operation

```

1 @cached
2 operation ECore!EClass isNode() : Boolean {
3
4   if (self.isLink()) return false;
5   if (self.abstract) return false;
6
7   var isNode := self.isAnnotatedAs('gmf.node');
8   var isNoNode := self.isAnnotatedAs('gmf.nonode');
9
10  if (isNoNode) return false;
11  else if (isNode) return true;
12  else return self.eSuperTypes.exists(s|s.isNode());
13
14  return isNode;
15 }

```

Another example of a user-defined constraint is the *IsGmfLink* constraint, which is illustrated in listing 6.7. This constraint checks whether the EClass which is referred by the EClass link end of an EClass2ConnectionTraceLink is annotated with the *@gmf.link* annotation. If this EClass is not annotated with this annotation, then it should not be linked to a Connection. In appendix E, the constraints used for this case study are provided.

Listing 6.7: The *IsGmfLink* constraint

```

1 context EClass2ConnectionTraceLink {
2   //The annotation link end must be a gmf.node
3   constraint IsGmfLink {
4
5     check : self.EClass.isLink()
6
7     message : "EClass " + self.EClass.name + " is not annotated as
           @gmf.link"
8   }
9 }

```

By using the TML domain specific language, we were able to capture all the necessary traceability information for this scenario in a scenario-specific metamodel. Moreover, we were able to capture the semantics of scenario-specific correctness constraints.

6.1.3.5 Establishing traceability

In the next step, we establish the scenario specific traceability information. This is achieved by following the approach described in section 4.2. In this scenario, the filesystem Ecore metamodel with the accompanying EuGENia-specific annotations is used to generate the GMF models, required to produce the filesystem visual editor. To produce the intermediate GMF models a model-to-model transformation (Ecore2GMF) is used. This transformation is expressed in EOL and it contains 1167 lines of code. The entire transformation for EuGENia can be found at the Epsilon subversion repository:

<http://dev.eclipse.org/svnroot/modeling/org.eclipse.gmt.epsilon>

Using EOL as the transformation language means that no internal trace is produced when the transformation is executed. Therefore, in order to generate the *GTrace* model for this particular scenario, we have to inject traceability specific code in the transformation specification. This statements have the generic format illustrated in listing 6.8. This statement uses the *add* method (its implementation is illustrated in listing 4.15) to create a *GTraceLink* for the various relationships, which are expressed in the transformation.

Listing 6.8: EOL statement for producing *GTraceLinks*

```

1 trace.add(source, target, "");

```

Once the appropriate code is injected in the EOL transformation specification, we can execute the EuGENia transformation and generate the intermediate GMF models and the traceability model for this scenario. Part of this model is illustrated in figure 6.6.

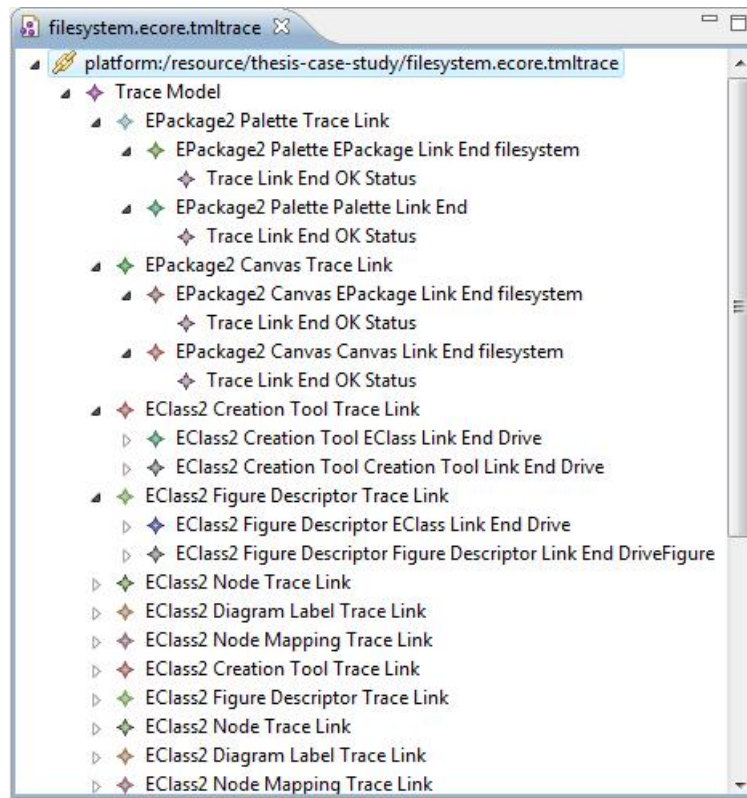


Figure 6.6: Illustration of the filesystem traceability model

The final step for creating the traceability model is to execute the correctness constraints, which accompany the traceability metamodel in order to validate that the produced traceability model is valid. Now that the traceability model is produced, it can be used during the development lifecycle and it can be maintained and evolved if the models referenced by the traceability metamodel change.

By using the proposed approach, we were able to semi-automatically generate the traceability metamodel. If ETL had been used as the transformation language, the computation of the traceability metamodel would have been automatic.

6.1.3.6 Using and maintaining traceability

In this section, we will present how the produced traceability model can be used to keep consistent the filesystem Ecore model with its related GMF models, when the filesystem model is refactored. Moreover, we will show how TML can be used to maintain the integrity of the establish traceability model. To achieve this, we use the approach proposed in sections 4.4.2 and 4.3.

In order to propagate any possible changes of the filesystem model to the GMF models appropriate model wizards have been developed. These wizards use the case-specific traceability information to identify which model elements are related to the modified one. For the purposes of this case study, we have developed three such wizards. The first one is illustrated in listing 6.9. This wizard renames an EClass, when a *gmf.node* annotation is attached to it. In line 3 a guard is specified, which uses the *isNode* user-defined operation. This guard guarantees that this wizard is applied only to EClasses, which have the *gmf.node* annotation attached to them. In the *do* part of the wizard its behaviour is specified. When this wizard is executed, the user is prompted to provide a new name for the EClass. This is specified in line 9. In line 12, the appropriate GMF models and the corresponding traceability model are loaded into memory. In the rest of the body of the wizard, the traceability information is used to update elements of the GMF models, which are related to the renamed EClass. In lines 15 - 17 instances of the *FigureDescriptor* metaclass, which are related to the renamed element via an *EClass2FigureDescriptorTraceLink*, are renamed accordingly. In a similar manner, the name of the instances of the *Node*, *DiagramLabel* and *CreationTool* metaclasses is changed to the new name of the EClass. Finally, in line 35 the user-defined name is assigned to the *name* attribute of the EClass.

Listing 6.9: The *RenameNodeClass* wizard

```

1  wizard RenameNodeClass {
2
3  guard : self.isNode()
4
5  title : 'Rename class'
6
7  do {
8
9  var newName = UserInput.prompt("New name", "Select a new name");
10
11 var modelLoader = new Native("org.eclipse.epsilon.tml.eugenia.
    refactoring.GmfModelLoader");
12 modelLoader.loadOtherModels(self);
13
14 // Update respective FigureDescriptor
15 for (e2fd in Trace!EClass2FigureDescriptorTraceLink.all.select(
    e2fd|e2fd.EClass.target = self)) {
16     e2fd.FigureDescriptor.target.name = newName + "Figure";
17 }
18
19 // Update respective Node

```

```

20  for (e2n in Trace!EClass2NodeTraceLink.all.select (e2n|e2n.EClass.
    target = self)) {
21    e2n.Node.target.name = newName;
22  }
23
24  // Update respective DiagramLabel
25  for (e2dl in Trace!EClass2DiagramLabelTraceLink.all.select (e2dl|
    e2dl.EClass.target = self)) {
26    e2dl.DiagramLabel.target.name = newName + "Label";
27  }
28
29  // Update respective Tool
30  for (e2t in Trace!EClass2CreationToolTraceLink.all.select (e2t|e2t.
    EClass.target = self)) {
31    e2t.CreationTool.target.`title` = newName;
32    e2t.CreationTool.target.description = "Create new " + newName;
33  }
34
35  self.name = newName;
36 }
37 }

```

Similarly, the *RenameLinkClass* wizard illustrated in listing 6.10, renames an EClass, which is annotated with the *gmf.link* annotation. In line 3 the appropriate guard is defined in order to ensure that this wizard applies only to EClasses with the aforementioned annotation attached to them. The rest of the body of this wizard is similar to the *RenameNodeClass* wizard defined above.

Once the appropriate wizards are defined, the user can choose the *Rename* menu option and rename simultaneously the EClass and its associated model elements in the GMF models. An illustration of this menu is shown in figure 6.7.

When the name of an EClass changes the validity of the traceability model is not violated, since the relevant trace links point to valid model elements. What needs to be updated though is the value of the *Name* maintenance data attribute which is attached to the EClass link end of the relevant trace links. This value can be updated by selecting the “*Maintain Trace*” option of the context specific menu of the traceability model.

Listing 6.10: The *RenameLinkClass* wizard

```

1  wizard RenameLinkClass {
2
3  guard : self.isLink()
4

```

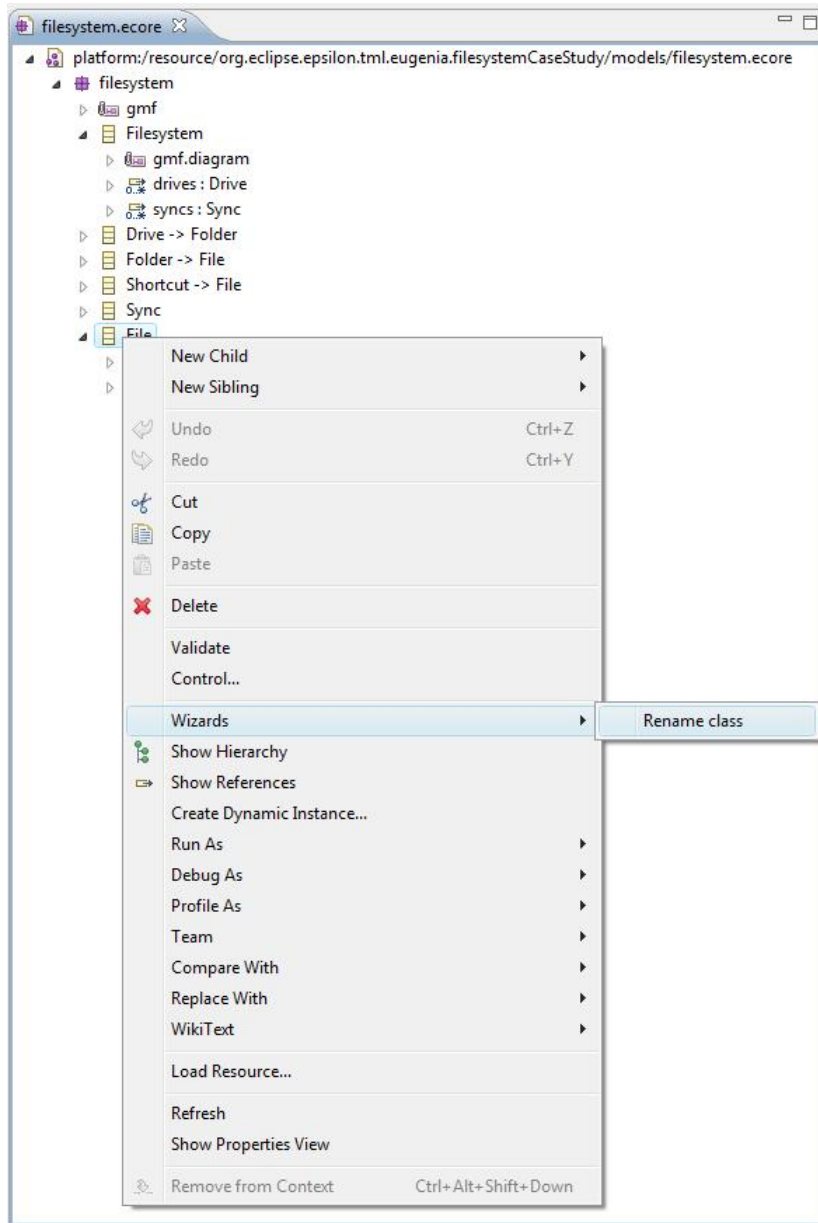


Figure 6.7: Illustration of the *RenameClass* menu

```

5  title : 'Rename class'
6
7  do {
8      var newName = UserInput.prompt("New name", "Select a new name");
9
10     var modelLoader = new Native("org.eclipse.epsilon.tml.eugenia.
        refactoring.GmfModelLoader");
11     modelLoader.loadOtherModels(self);
12
13     // Update respective FigureDescriptor
14     for (e2fd in Trace!EClass2FigureDescriptorTraceLink.all.select(
        e2fd|e2fd.EClass.target = self)) {
15         e2fd.FigureDescriptor.target.name = newName + "Figure";
16     }
17
18     // Update respective Connection
19     for (e2c in Trace!EClass2ConnectionTraceLink.all.select(e2c|e2c.
        EClass.target = self)) {
20         e2c.Connection.target.name = newName;
21     }
22
23     // Update respective Tool
24     for (e2t in Trace!EClass2CreationToolTraceLink.all.select(e2t|e2t.
        EClass.target = self)) {
25         e2t.CreationTool.target.`title` = newName;
26         e2t.CreationTool.target.description = "Create new " + newName;
27     }
28
29     self.name = newName;
30 }
31 }

```

The third wizard is the *DeleteClass* wizard illustrated in listing 6.11. When this wizard is executed it deletes an EClass and all the other elements in the GMF models that are related to this EClass. The structure of this wizard is similar to the structure of the ones specified above. The first part of the wizard consists of a guard. This guard specifies that this wizard applies to EClasses, which are not *eSuperTypes* of any other EClass. In the *do* part of the wizard, the relevant GMF models and the traceability model are loaded into the memory. Finally, the elements of the GMF models, which are related to the deleted EClass via various trace links are deleted as well.

Listing 6.11: The *DeleteClass* wizard

```
1 wizard DeleteClass {
2
3   guard : not EClass.all.exists(c|c.eSuperTypes.includes(self))
4
5   title : 'Delete class'
6
7   do {
8     var modelLoader = new Native("org.eclipse.epsilon.tml.eugenia.
          refactoring.GmfModelLoader");
9     modelLoader.loadOtherModels(self);
10
11    // Update respective FigureDescriptor
12    for (e2fd in Trace!EClass2FigureDescriptorTraceLink.all.select(
          e2fd|e2fd.EClass.target = self).clone()) {
13      delete e2fd.FigureDescriptor.target;
14      delete e2fd;
15    }
16
17    // Update respective Node
18    for (e2n in Trace!EClass2NodeTraceLink.all.select(e2n|e2n.EClass.
          target = self).clone()) {
19      delete e2n.Node.target;
20      delete e2n;
21    }
22
23    // Update respective DiagramLabel
24    for (e2dl in Trace!EClass2DiagramLabelTraceLink.all.select(e2dl|
          e2dl.EClass.target = self).clone()) {
25      delete e2dl.DiagramLabel.target;
26      delete e2dl;
27    }
28
29    // Update respective Tool
30    for (e2t in Trace!EClass2CreationToolTraceLink.all.select(e2t|e2t.
          EClass.target = self).clone()) {
31      delete e2t.CreationTool.target;
32      delete e2t;
33    }
34
35    delete self;
```

36 }

37 }

Deleting the link ends of the trace links, which point to a deleted EClass affects the validity of the traceability model, since these links have dangling link ends. In this case-study, we have implemented the wizard in such a way, that these links are deleted when the wizard is executed. If the wizard was not implemented in this manner, then we could have used the “*Maintain Trace*” option, in order to modify the traceability model and reestablish its validity by deleting the trace links with dangling link ends.

By using the proposed approach, we were able to develop wizards which propagate changes to affected models. Moreover, we were able to maintain the traceability model, when such changes took place.

6.2 Evaluation of the contributions

In this section, the contributions of this thesis are examined individually. The discussion that follows is based on the analysis of the application through the means of evaluation described above.

6.2.1 Classification of Traceability Approaches (Section 3.2)

In chapter 3, a classification of traceability approaches has been developed using the technique of numerical taxonomy. This classification is unique and novel in the sense that the existing traceability classifications are informal in nature and none of them refers to, or applies, any concepts or techniques from the science of classification.

The contribution in this area is twofold. First, the application of numerical taxonomy in the domain of traceability is novel and it led to the construction of a well defined theoretical classification of software traceability approaches. The second contribution is the produced classification per se. This outcome is captured in the form of the dendrogram illustrated in figure 3.1. This tree structure represents the overall similarity of the traceability approaches used in this study based on their characteristics and it can provide a system for conducting, documenting and coordinating a comparative study of the various traceability approaches.

By using the dendrogram, we were able to compare the various traceability approaches based on their overall characteristics without having to choose which characteristics might be of interest. These characteristics reflected design choices, which affect the applicability of an approach to different scenarios. The data matrix, which accompanies the dendrogram makes explicit the possible design choices for the traceability approaches under study, which is one of the main contributions of this thesis.

Furthermore, based on the clustering results we have identified a few major categories in which most approaches fit.

Finally, the produced classification proved also a very successful means of reasoning about the need of the proposed approach and positive comments as well as further interest about the application of numerical taxonomy in different scenarios have been expressed by research peers.

6.2.2 Traceability Metamodelling Language (Section 4.1)

In section 4.1, TML was presented. The purpose of TML is to enable the construction of case-specific and semantically-rich traceability metamodels as well as the generation of their accompanying correctness constraints in order to provide model-to-model traceability support in MDE processes. The motivation for TML was based on the existing traceability literature (e.g. (Aizenbud-Reshef *et al.*, 2006a; Paige *et al.*, 2008; Walderhaug *et al.*, 2008)), which argues that it is necessary to define different types of trace links for different application scenarios. Moreover, researchers argue that traceability models should be well-defined so that they can be automatically manipulated by software tools (e.g (Aizenbud-Reshef *et al.*, 2006a; Walderhaug *et al.*, 2008)).

By experimenting with TML in different traceability scenarios, we determined that TML fulfills its requirements. One can define easily traceability metamodels with scenario-specific semantics and a set of accompanying constraints. The traceability models which result from using TML are well-defined in the MDE sense (i.e. they conform to a well-structured metamodel and they are accompanied by correctness constraints). Therefore, they can be automatically manipulated by using existing model management frameworks.

The concepts which comprise TML have been defined in an iterative manner using the approach in different traceability scenarios and examples. The application of the proposed approach in different scenarios enabled the constant evaluation of the correctness and completeness of TML. Moreover, the use of TML in the examples and the case-study demonstrates the benefits of using the proposed approach.

Another benefit of TML is the fact that it uses the existing technologies in its application domain, i.e. the MDE domain. TML is defined using metamodelling and model management tasks such as model transformation, model validation, etc. As a result the learning curve for a developer, who is familiar with MDE, is not steep. Therefore it is reasonable anticipated that TML can be extended or adjusted/modified by the MDE community with minimal effort.

6.2.3 Traceability Identification with TML (Section 4.2)

The main contribution of section 4.2 is an approach for the automatic computation of traceability models. Traceability models can be created manually by using a dedicated tree editor, but generating them automatically requires less effort and it is less error-prone. The proposed approach uses model management operations for the computation of traceability models, and then enhances these models with case-specific semantics using TML models.

One of the benefits of the proposed solution to the computation of traceability models is the fact that it is framework-agnostic. The reference implementation uses the Epsilon model management framework, but any other framework could have been used instead as long as an internal trace is generated while executing model management tasks. The genericity of the proposed approach can be attributed to the addition of an abstraction level between TML and the model management framework, which is enhanced with TML. This abstraction level hides the differences and particularities of the different model management frameworks.

An important characteristic of the proposed approach is the fact that it is model-centric. This means that for the computation of the traceability models both the structural metadata as well as the lexical information of models are used. Structural metadata can be cardinalities of model elements or their associations or possibly different types of containments. This is achieved by using existing model management languages, which are built for this task.

Using the identification approach, we were able to create traceability models in different application scenarios in an automatic manner. One limitation though of the approach is that it can not generate automatically informal aspects of a traceability model. These informal information are captured in dedicated model elements called *Contexts* in TML. In the proposed approach the contexts have to be filled in manually by the developer once the traceability model is created.

In section 3.3, we argued why a traceability approach should support the two viewpoints of traceability: Dependency and Generative. The application of the proposed approach to the different traceability scenarios has demonstrated that it supports both traceability viewpoints. By using the trace produced by transformation languages a scenario-specific traceability model can be computed automatically, covering the Generative viewpoint. Moreover, by comparing different models using a comparison language and then utilising the trace produced by the comparison language can compute a traceability model, covering the Dependency viewpoint.

In conclusion, the evaluation of the proposed approach to the computation of traceability models has been successful. The means of evaluation have been applied extensively throughout the research and provided sufficient evidence for the feasibility and applicability of the proposed approach. This approach fulfills all the identified requirements for a trace link recovery method in the context of MDE, as they were identified

in section 4.2.

6.2.4 Traceability Maintenance with TML (Section 4.3)

The main contribution of section 4.3 is an approach for the semi-automatic maintenance of traceability models. The integrity and validity of traceability models can be violated, when the models they refer to change. This thesis proposes a novel approach to maintaining the validity of traceability models by storing dedicated maintenance metadata, as well as expressions to calculate them. When a the validity of a link is considered to be violated, then a maintenance algorithm is executed in order to reconcile the broken link. If the automatic reconciliation of the link is not possible then user-input is required.

By using this approach in the various scenarios, we found out that it can detect broken links in the majority of the cases. Therefore, even in the cases where the automatic reconciliation was not feasible, the broken links are reported back to the user for manual reconciliation. As a result, the traceability model is in a valid state. During the application of the approach, we have noticed that the results of the maintenance depend heavily on the definition of the appropriate maintenance data. This implies that the nature of the maintenance data should be defined by someone who has deep understanding of the metamodels between which traceability is captured.

Furthermore, it was found that the proposed approach was able to support link maintenance for all model change types. This is due to the fact that it is a state-based method, which does not rely on identifying model changes. As a result, even in complex scenarios where the compound changes can not be easily identified, the proposed approach can find broken links and reconcile them.

Overall, the approach to traceability maintenance proposed in this thesis is found to be novel and beneficial for traceability support.

6.2.5 Traceability Usage (Section 4.4)

Section 4.4 discusses two novel uses of scenario-specific traceability in the context of MDE. In the first usage-scenario, traceability models, which are computed during the execution of a transformation are used to identify possibly problematic sections of transformations. This sections can be then tested extensively using various transformation testing techniques. In the second usage scenario, traceability can be used to propagate changes and keep models consistent during model refactorings.

Both usage scenarios were applied to a variety of examples. The evaluation of the first application scenario (i.e. the use of traceability for testing transformations) was positive. Using the proposed approach a number of transformation specification defects were identified such as wrong cardinalities or transformation rules which transformed wrong model elements. The evaluation of the second usage scenario was positive as

well. Changes were successfully propagated after model refactorings. During our experimentation, we found that the types and complexity of changes which can be propagated to other models depend solely on the expressive power of the transformation language, which is used to specify the refactorings. The combination of EWL and EOL proved to be expressive enough to capture and propagate even complex refactorings.

6.3 Evaluation of the Thesis Proposition

The contributions were also assessed in terms of the distinct characteristics of the thesis proposition, which were identified in section 3.4.

1. **Model-based Approach:** Traceability information is captured as a model which conforms to a well-defined traceability metamodel. Moreover, model management tasks and operations are used to create, modify and update the traceability models. The proposed approach provides traceability support for modelling artefacts taking into consideration specific characteristics, such as the structure of the model. Therefore, we conclude that the proposed approach is model-based.
2. **Manage Traceability:** The proposed approach provides support for all four traceability activities, i.e. representation, identification, maintenance and usage of traceability. TML can be used to define well-structured, scenario-specific traceability metamodels. Moreover, it can be used to generate additional correctness constraints, which accompany the traceability metamodel. The approach proposed in section 4.2 can be used to support the computation of traceability models, which conform to a well-defined traceability metamodel. Furthermore, the approach presented in section 4.3 can be used to maintain the integrity of a traceability model, when the models it refers to change. Finally, section 4.4 proposes two novel usage scenarios for traceability information.
3. **Heterogeneous Traceability Relationships:** Using the proposed approach, an engineer is able to manage traceability relationships between models expressed in heterogeneous languages. Since the proposed approach does not depend on a particular technology, it can support different languages and modelling technologies, if the appropriate infrastructure is developed. The concepts described and used in this thesis are language-agnostic and should apply to different traceability scenarios.
4. **Rigorous:** The proposed approach supports the capture of case-specific or scenario-specific semantics. This is achieved by enabling the developer to define a different case-specific traceability metamodel with its accompanying constraints for different scenarios with reduced effort.

5. **Automation:** The proposed approach provides semi-automatic identification and maintenance of traceability information. This is achieved by using model-management frameworks in order to manipulate the well-defined traceability models.
6. **Derived and Initial Models:** The proposed approach is suitable for traceability between models, which are generated automatically by applying model operations on other models (derived models) as well as between models, which are created manually by engineers (initial models). This is achieved by utilising the traces produced from both transformation and matching model management tasks and enhancing them with scenario-specific semantics.

6.4 Shortcomings and Limitations

This section presents the main limitations and shortcomings of the proposed approach and reference implementation.

6.4.1 Lack of support for non-model artefacts

The approach proposed in this thesis focuses on providing traceability support between models. However, when applying MDE, development of models starts from other kinds of artefacts such as informal, natural language descriptions of requirements or spreadsheets and it ends usually with textual artefacts such as executable source code and its accompanying documentation. Following (Paige *et al.*, 2008), a traceability approach should consider how models can be traced to other (nonmodel) artefacts and how (non-model) artefacts can be traced to models. Therefore, in order to be able to support end-to-end traceability in MDE - that is, traceability between all artefacts developed and generated in an MDE systems development process - then we need to extend our approach to support non-model artefacts.

6.4.2 Lack of support for custom traceability information

Another limitation of the approach described in this thesis is the lack of support for custom traceability information. In TML, developers can capture custom information in dedicated metaclasses called *Contexts*. These metaclasses can be used to capture concepts, which are not supported by the TML metamodel. An example of such information is the name of a developer, who is responsible for maintaining a particular trace link. Although TML gives the ability to developers to capture this kind of information, when computing the traceability models, the proposed approach is not able to generate this information automatically. The different contexts, which are attached to the various

links of the traceability model, have to be filled in manually by the developer. That is why we consider our approach to provide semi-automatic identification, instead of automatic.

6.5 Chapter Summary

This chapter has confirmed the feasibility and validity of the thesis hypothesis by using several means of evaluation. First and foremost, the use of the reference implementation in a complex traceability case-study. Moreover, the approach was applied to smaller examples and case-studies throughout the duration of this research. Finally, publications and acceptance by the research community has provided valuable feedback and confidence for the validity of the TML approach.

Overall, the evaluation has demonstrated that it is feasible to develop an approach, which fulfills the requirements identified in section 3.4. Furthermore, the application of the proposed approach to support traceability in MDE can be beneficial in terms of applicability, automation and rigorousness. Finally, the evaluation has identified limitations and shortcomings of the approach such as the lack of support for custom information and the lack of support for non-model artefacts.

Chapter 7

Conclusions and Future Work

This thesis has presented an integrated approach for defining and managing case-specific, model-to-model traceability information in a rigorous manner. The contributions of this thesis can be summarised as follows:

- Understand and analyse existing literature on traceability by developing a classification using numerical taxonomy (section 3.2).
- Specification of TML, which enables the construction of case-specific and semantically-rich traceability metamodels as well as the generation of their accompanying correctness constraints (section 4.1).
- Support for semi-automatic identification of traceability information between models (section 4.2).
- Support for semi-automatic maintenance of the validity of traceability information between models (section 4.3).
- Usage of traceability information for transformation testing (section 4.4.1)
- Usage of traceability information for change propagation in model refactoring scenarios (section 4.4.2)

These contributions support the thesis proposition stated in section 3.4:

This thesis demonstrates that a domain specific model-based traceability approach can support and automate the process of rigorously managing the different types of heterogeneous traceability relationships between both derived and initial models in an MDE process.

Moreover, the thesis contributions satisfy the thesis objectives, which were the following:

- To propose an approach with which rigorous, well-defined, case-specific traceability models can be developed.
- To support and automate the activity of identifying traceability links between models.
- To support and automate the activity of maintaining traceability information between models.
- To propose novel usage scenarios of traceability information in MDE.

This chapter discusses how the thesis proposition is supported by the above contributions. Moreover, the conclusions and findings of this thesis are summarised and areas of further work are identified.

7.1 Review Findings

In chapter 2 a review of the existing work in the field of software traceability was performed. During the review, the main traceability approaches and their characteristics were identified. These findings guided the analysis, which was conducted in chapter 3. In this analysis, the phenetic approach or else numerical taxonomy was followed. The result of this analysis is the data matrix shown in table B.1 and the dendrogram illustrated in figure 3.1. By analysing the outcomes of the phenetic process, we made a set of observations.

First, the majority of the traceability approaches included in the analysis deal with only a subset of the traceability activities. As a result, they are defined rather isolated, using different, not necessarily combinable techniques and technologies. This consequently makes their integration into a comprehensive traceability environment a challenge.

Moreover, traceability approaches tend to focus on only one viewpoint of traceability ignoring the other one. Considering that in MDE processes, both viewpoints (Dependency and Generative) are present, we considered this as a main limitation for applying one of the existing traceability approaches to an MDE process.

A third observation, was the fact that the approaches, which automate the management of traceability information, tend not to define semantics in an enforceable way. As a result, limited support for rich analysis of traceability information can be provided. Furthermore, we observed that many of the existing approaches focus only on particular

languages and artefact types limiting their applicability to specific traceability scenarios. Finally, we observed that very few approaches consider the activity of traceability maintenance.

Based on the above observations, we specified a set of requirements, which must be fulfilled by a traceability approach in the context of MDE and the thesis proposition was stated in section 3.4.

7.2 Proposed Solution

In chapter 4 the main knowledge contributions of this thesis are imparted. These contributions are directly related to the four aspects of traceability ; namely representation, identification, maintenance and usage. In the following sections, we will summarise the results of this thesis.

7.2.1 Traceability Metamodelling Language

The main contribution of this research work is the specification of TML, which is presented in section 4.1. One of our observations during the review of the literature was the fact that scenario-specific traceability information with rich semantics is rarely supported by existing approaches, especially the ones whose focus is on automating the computation of traceability models or on automating the maintenance of the captured trace links. In order to provide support for specifying such traceability models - i.e. traceability models with scenario-specific semantics - we developed TML.

TML is a domain specific metamodelling language, whose purpose is to enable the construction of scenario-specific traceability metamodels and their accompanying constraints with reduced effort. Therefore, TML facilitates the specification of traceability domain specific languages, each of them targeting a concrete instance of the software traceability domain. There are three distinct reasons, which motivate the need for having a dedicated traceability metamodelling language. First, context-specific trace link types are needed. A generic language will invariably use generic trace link types. Therefore, rich analysis of traceability information is not supported, since the intended meaning of the trace links is not precisely captured. Furthermore, context-specific constraints should be defined in order to capture the intended meaning of trace links - i.e. the semantics of trace links - more accurately. Finally, domain-specific functionality should be reused across the different instances of the language. For example, supporting the automatic computation of traceability models should be provided for the different scenario-specific traceability languages.

By using TML, we achieve systematic reuse of common traceability concepts and infrastructure elements such as trace link constraints or maintenance algorithms across

the different case-specific traceability languages. Moreover, we achieve to systematically support the definition of variability between the scenario-specific traceability languages. This is achieved by specifying a well-defined set of variation points such as trace link types or trace link ends. These variation points are supported consistently in the different parts of the infrastructure, which accompanies TML.

7.2.2 Traceability Identification with TML

One of the objectives of this thesis was to automate the computation of traceability models. In section 4.2, an approach to the computation of scenario-specific traceability models is presented. The motivation for developing a dedicated approach for the computation of traceability models has to do with the fact that manual creation of such models is time consuming, labour intensive and error prone.

The proposed approach uses model management operations for the computation of traceability models, and then enhances these models with case-specific semantics using TML models. Therefore, we are reusing part of the existing infrastructure of the model management frameworks, rather than developing this infrastructure from scratch. Although the proposed solution is tightly coupled with the existence of a model management framework, it is framework-agnostic. The provided reference implementation uses the Epsilon model management framework, but any other framework could have been used instead as long as an internal trace is generated while executing model management tasks. The genericity of the proposed approach can be attributed to the addition of an abstraction level above TML. This abstraction level hides the differences and particularities of the different model management frameworks.

The resultant approach is a model-centric approach, which can support both traceability viewpoints; namely the Dependency and the Generative viewpoints. To support the Generative viewpoint, the internal trace of a transformation language such as ETL is enhanced with scenario-specific semantics, which are derived from a TML model. On the other hand, to support the Dependency viewpoint, a model matching language such as ECL is used to find similarities between models, and its internal trace is enhanced with scenario-specific semantics in order to compute the traceability model.

7.2.3 Traceability Maintenance with TML

In section 4.3 an approach to the semi-automatic maintenance of traceability models is presented. Traceability maintenance is one of the four main activities associated with software traceability and it is closely related to software entropy. Its aim is to prevent traceability models from degrading as the artefacts they refer to are modified. As discussed in section 2.4.3, there are two main approaches to traceability maintenance; namely event-driven and state-based approaches. Both of them have advantages and dis-

advantages. However, in the context of this thesis we propose a state-based approach, since such an approach is more suitable for the traceability scenarios we are considering in this research work. In such scenarios, we provide traceability support for different and possibly heterogeneous notations. Event-driven approaches on the other hand are usually better suited for notation-specific approaches, since they restrict their application domain in order to be able to identify complex and compound model changes. Therefore, the proposed approach to traceability maintenance is a state-based approach. This thesis proposes the use of dedicated maintenance metadata and accompanying expressions to maintain the validity of traceability models.

7.2.4 Traceability Usage

In section 4.4 two novel traceability usage scenarios in the context of MDE were presented. In the first usage-scenario, traceability models, which are computed during the execution of a transformation, are used to identify possibly problematic sections of this transformation. These sections can then be tested extensively using various transformation testing techniques. The proposed approach relies on the internal trace produced by transformation engines to identify problematic transformation rules in a transformation specification. If a trace is generated during the transformation execution, which does not conform to the scenario-specific traceability metamodel, then this rule should be further tested.

In the second usage scenario, traceability can be used to propagate changes and keep consistent models during model refactorings. As developers modify or refactor development entities such as models in order to improve in a disciplined way some of their qualitative attributes, they must ensure to update other system models in order to be consistent with these changes. Therefore, in this thesis we propose the combination of a task specific language designed for specifying model refactorings with TML models in order to propagate model changes.

7.3 Evaluation Results

In chapter 6, the validity of the thesis proposition presented in section 3.4 was confirmed. Different means of evaluation were employed during the different stages of the research. In section 6.1.1, evaluation of the various concepts expressed in this thesis was performed by using small scale examples and case studies. In section 6.1.2, the impact of the proposed approach was assessed in terms of publicity and external references. Finally, in section 6.1.3, the thesis proposition was evaluated in terms of feasibility and potential benefits through a complex case study.

7.4 Areas of Further Work

Several directions to further work have been identified as a result of this work. One direction is to investigate the need for traceability support of non-model artefacts for MDE processes. Another possible research direction is to investigate how the concept of domain-specific metamodelling language (DSM2L) can be applied in different scenarios of families of languages.

7.4.1 Support for non-model artefacts

Although models are considered to be the main type of artefact used in MDE, there are other kinds of artefacts which are extensively used. For example, informal, natural language descriptions of requirements and spreadsheets are used during the initial phases of the development lifecycle. Moreover, source code and the system's documentation are produced at later stages. Therefore, a comprehensive approach to traceability support for MDE needs to consider these types of artefacts as well. It should artefacts as well, in terms of how models can be traced to other (non-model) artefacts and how (non-model) artefacts can be traced to models.

One issue which arises with textual artefacts is the fact that it is not trivial to uniquely identify pieces of text, when they are part of a larger document. Currently, naive offset/length-based solutions are used, but those are sensitive to subsequent text editing actions. A possible solution to this problem could be the use of statistical techniques from the area of Natural Language Processing in order to facilitate referencing.

7.4.2 Expand the concept of DSM2L

This work also raises issues to be investigated in a broader context. Through this work, a novel approach to developing traceability domain-specific languages has been presented. This approach was based on the concept of a domain-specific metamodelling language.

Families of languages similar to case-specific traceability languages appear in other domains of software engineering such as model differencing or Software Product Lines (SPL). In such families, there are common recurring patterns and variations across the different languages. It would be of interest to investigate the challenges associated with the development of such families of languages. Moreover, it would be interesting to investigate whether an approach similar to the one followed in this thesis would be beneficial for the development of families of domain-specific languages.

Appendix A

Characters Used in the Phenetic Analysis

In this appendix, the characters used in the phenetic analysis are presented and a brief description for each one of them is provided. These characters are the following:

Manual Identification of Links: Previously unknown relationships between artefacts are discovered and recored manually.

Semi-automatic Identification of Links: Previously unknown relationships between artefacts are discovered and recored in a partly automatic manner. This means that the identification process is automatic up to point, but user input is required.

Automatic Identification of Links: Previously unknown relationships between artefacts are discovered and recored automatically.

Identification with IR techniques - VSM: The identification of previously unknown relationships between artefacts is done by using the Vector Space Model to estimate the similarity between two entities.

Identification with IR techniques - LSI: The identification of previously unknown relationships between artefacts is done by using the Latent Semantic Indexing method to estimate the similarity between two entities.

Identification with IR techniques - PM: The identification of previously unknown relationships between artefacts is done by using probabilistic models to estimate the similarity between two entities.

Identification with indirect rules: Identification of previously unknown relationships between artefacts is performed by utilising rules, which discover implicit relationships between entities.

Identification with direct rules: Identification of previously unknown relationships between artefacts is performed by utilising rules, which discover explicit relationships between entities.

Identification with program analysis: Previously unknown relationships between artefacts and source code are discovered by using static code analysis.

Identification with run-time monitoring: Previously unknown relationships between artefacts and source code are discovered by using dynamic code analysis.

Identification augmented with A.I. techniques: The identification method is enhanced with artificial intelligence methods, such as machine learning.

Identification augmented with visualisation techniques: The identification method is enhanced with visualisation methods, such as link colouring.

Identification with history analysis: Previously unknown relationships between artefacts are discovered by identifying patterns of change in a repository.

Implicit identification with transformations: Previously unknown relationships between artefacts are discovered by using information provided by a transformation engine.

Explicit identification with transformations: Previously unknown relationships between artefacts are discovered by using traceability code inserted in transformation code.

Guidance for traceability usage: The traceability approach proposes how captured traceability can be used.

Inter-artefact storage of traceability information: traceability information is stored separately from the artefacts it refers to.

Intra-artefact storage of traceability information: traceability information is stored in the artefacts it refers to.

Case-specific metamodel- extensible: The semantics captured by the traceability metamodel can be manipulated in order to support different traceability scenarios.

Generic traceability metamodel: The semantics captured by the traceability metamodel are generic and they do not support specific traceability scenarios.

Representation with hyperlinks: Captured traceability information is represented using a hyperlink system.

Representation with graphs: Captured traceability information is represented using graph structures.

Representation with inline tags: Captured traceability information is represented tags in the artefacts the traceability refers to.

Representation with matrices: Captured traceability information is represented using matrix structures.

Representation with hyperlinks: Captured traceability information is represented using a hyperlink system.

Semantically-rich link semantics: Traceability semantics is captured in a rigorous manner using case-specific metamodels as well as case-specific correctness constraints.

Text-to-text traceability: The approach supports traceability information between textual artefacts.

Model-to-text traceability: The approach supports traceability information between models and textual artefacts.

Model-to-model traceability: The approach supports traceability information between models.

Artefact-specific support: The approach supports traceability between specific artefacts (e.g. between UML class diagrams and source code).

Artefact-agnostic support: The traceability approach can be applied to various artefacts.

State-based maintenance: The maintenance of traceability information is performed by comparing different states of the traceable artefacts.

Event-driven maintenance: The maintenance of traceability information is triggered by particular events, which affect the traceable artefacts.

Automatic traceability maintenance: Traceability information is maintained automatically.

Manual traceability maintenance: Traceability information is maintained manually.

Semi-automatic traceability maintenance: Traceability information maintenance is performed in a partly automatic way, requiring user input.

Dependency viewpoint support: The approach supports traceability between existing artefacts.

Generative viewpoint support: The approach supports traceability between a generated artefact and the one which was used for its generation.

Appendix B

Phenetic Analysis Data

In this appendix, the data used in the phenetic analysis is presented. Table B.1 illustrates the different states of a character, while table B.2 shows the state possessed by each character for the various OTUs.

Table B.1: List of Character States

#	Character	Character State
1	Manual Identification of Links	0 = Identification is not performed manually 1 = Identification is performed manually
2	Semi-automatic Identification of Links	0 = Identification is not performed semi-automatically 1 = Identification is not performed semi-automatically
3	Automatic Identification of Links	0 = Identification is not performed automatically 1 = Identification is not performed automatically
4	Identification with IR techniques - VSM	0 = Identification with IR (VSM) techniques is not supported 1 = Identification with IR (VSM) techniques is not supported
5	Identification with IR techniques - LSI	0 = Identification with IR (LSI) techniques is not supported 1 = Identification with IR (LSI) techniques is not supported

Continued on next page

Table B.1 – continued from previous page

#	Character	Character State
6	Identification with IR techniques - PM	0 = Identification with IR (PM) techniques is not supported 1 = Identification with IR (PM) techniques is not supported
7	Identification with indirect rules	0 = Rule-based identification with indirect rules not supported 1 = Rule-based identification with indirect rules supported
8	Identification with direct rules	0 = Rule-based identification with direct rules not supported 1 = Rule-based identification with direct rules supported
9	Identification with program analysis	0 = Identification is not performed using program analysis 1 = Identification is performed using program analysis
10	Identification with run-time monitoring	0 = Identification is not performed using run-time monitoring 1 = Identification is performed using run-time monitoring
11	Identification augmented with A.I. techniques	0 = Identification is not augmented with A.I. techniques 1 = Identification is augmented with A.I. techniques
12	Identification augmented with visualisation techniques	0 = Identification is not augmented with vizualisation techniques 1 = Identification is augmented with vizualization techniques
13	Identification with history analysis	0 = Identification is not performed using history analysis 1 = Identification is performed using history analysis

Continued on next page

Table B.1 – continued from previous page

#	Character	Character State
14	Implicit identification with transformations	<p>0= Identification is not based on injecting code in transformation code</p> <p>1= Identification is based on injecting code in transformation code</p>
15	Explicit identification with transformations	<p>0= Identification is not performed using information provided by a transformation engine</p> <p>1= Identification is performed using information provided by a transformation engine</p>
16	Inter-artefact storage of traceability information	<p>0= Traceability information is not stored externally to the artefacts it refers to</p> <p>1= Traceability information is stored externally to the artefacts it refers to</p>
17	Intra-artefact storage of traceability information	<p>0= Traceability information is not stored in the artefacts it refers to</p> <p>1= Traceability information is stored in the artefacts it refers to</p>
18	Case-specific traceability metamodel - extensible	<p>0= Traceability metamodel is not case-specific and extensible</p> <p>1= Traceability metamodel is case-specific and extensible</p>
19	Case-specific traceability metamodel - non extensible	<p>0= Traceability metamodel is not case-specific and non- extensible</p> <p>1= Traceability metamodel is case-specific and extensible</p>
20	Generic traceability metamodel	<p>0= Traceability metamodel is not generic</p> <p>1= Traceability metamodel is generic</p>
21	Representation with hyperlinks	<p>0= Traceability representation is not performed using hyperlinks</p> <p>1= Traceability metamodel is performed using hyperlinks</p>
22	Representation with graphs	<p>0= Traceability representation is not performed using graphs</p>

Continued on next page

Table B.1 – continued from previous page

#	Character	Character State
		1= Traceability metamodel is performed using graphs
23	Representation with inline tags	0= Traceability representation is not performed using inline tags 1= Traceability metamodel is performed using inline tags
24	Representation with matrices	0= Traceability representation is not performed using matrices 1= Traceability metamodel is performed using matrices
25	Semantically-rich link semantics	0= Trace link semantics are not defined rigorously 1= Trace link semantics are defined rigorously
26	Text-to-text traceability	0= Text-to-text traceability is not supported 1= Text-to-text traceability is supported
27	Model-to-text traceability	0= Model-to-text traceability is not supported 1= Model-to-text traceability is supported
28	Model-to-model traceability	0= Model-to-model traceability is not supported 1= Model-to-model traceability is supported
29	Artefact-specific support	0= Approach is artefact-agnostic 1= Approach is artefact specific
30	Artefact-agnostic Support	0= Approach is artefact-specific 1= Approach is artefact-agnostic
31	State-based maintenance	0= Traceability maintenance is not state-based 1= Traceability maintenance is state-based
32	Event-driven maintenance	0= Traceability maintenance is not event-driven 1= Traceability maintenance is event-driven
33	Automatic traceability maintenance	0= Traceability maintenance activity is not automatic 1= Traceability maintenance activity is automatic
34	Manual traceability maintenance	0= Traceability maintenance activity is not manual

Continued on next page

Table B.1 – continued from previous page

#	Character	Character State
		1= Traceability maintenance activity is manual
35	Semi-automatic traceability maintenance	0= Traceability maintenance activity is not semi-automatic 1= Traceability maintenance activity is semi-automatic
36	Guidance for traceability usage	0= Approach does not provide guidance on how to use traceability information 1= Approach provides guidance on how to use traceability information
37	Dependency viewpoint support	0= Dependency viewpoint is not supported 1= Dependency viewpoint is supported
38	Generative viewpoint support	0= Generative viewpoint is not supported 1= Generative viewpoint is supported

Table B.2: States Possessed by Each Character

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39				
Amar et.al 2008	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	1	0	0	1	0	0	0	0	0	1	0	1	1	0	0	0	0	0	0	0	1				
Antoniol et.al 2002	0	1	0	1	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	1	0	0	1	0	0	0	0	0	1	0	0	1	0				
Cleland-Huang et.al 2005	0	1	0	1	0	1	0	0	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	1	0	0	1	0	1	0	0	0	0	0	0	1	0				
Costa, M. & da Silva, A. R. 2007	1	0	1	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	1	0	0	0	0	0	0	0	1	0	1	1	0	1	1	0	0	0	1	1				
Egyed, A. & Grunbacher, P 2002.	0	1	0	0	0	0	1	1	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	1	0	0	1	0	1	0	0	0	1	0	1	1	0				
Fabro et.al 2005	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	1	0	0	0	0	0	1	0	0	1	0	0	1	0	1	1	0	0	0	0	0	1	0			
Falleri et.al 2006	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	1	0	0	1	0	0	1	0	0	0	1	0	1	1	0	0	0	0	0	0	0	1			
Grammel, B. & Voigt, K. 2009	0	0	1	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	0	0	1	1	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	1			
Grechanik et.al 2007	0	1	0	0	0	0	0	1	1	1	1	0	0	0	0	0	1	0	1	0	1	0	1	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	1	0		
Hayes et.al 2003	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	1	0	1	0	0	0	0	0	0	0	0	0	1	0		
Hayes et.al 2004	0	1	0	1	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	1	0	1	0	1	0	0	0	0	0	0	0	1	0		
Hayes et.al 2006	0	1	0	0	1	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	1	0	1	0	1	0	0	0	0	0	0	0	1	0		
Jirapanthong, W. & Zisman, A. 2007	0	0	1	0	0	0	1	1	0	0	0	0	0	0	0	0	1	0	1	0	1	0	1	0	0	0	1	1	0	0	1	0	1	0	0	0	0	0	0	1	0		
Jouault, F. 2005	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	1	0	1	0	1	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	1		
Kolovos, D. S. & Paige, R. F. 2010	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	1	0	1	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	1		
Lin et al. 2006	0	1	0	1	0	1	0	0	0	0	0	1	0	0	0	1	0	0	0	1	0	0	1	0	0	1	0	0	1	0	1	0	1	0	0	0	0	0	0	0	1	0	
Lucia et al. 2007 & 2008	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	1	0	0	0	1	1	0	0	0	1	0	0	1	0		
Mder, P. 2008	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	1	1	0	0	0	1	0	0	1	0		
Maletic et.al 2003	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	1	0	0	1	0	0	1	1	0	0	0	1	0	0	1	0	0	1	0	1	0	
Maletic et.al 2005	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	1	0	0	1	0	0	0	0	0	0	0	0	1	0		
Marcus, A. & Maletic, J. I. 2003	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	1	0	1	0	1	0	1	0	0	1	0	1	0	
Murta et.al 2006	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	1	0	0	1	0	0	0	0	0	0	1	0	0	1	0	
Natt och Dag et.al 2005	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	1	0	0	0	1	0	0	1	0	1	0		
Olsen, G. & Oldevik, J. 2007	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	1	1	0	0	0	1	0	0	1	0		
Pierce 1978	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	1	1	0	1	0	0	1	1	0	1	0	1	0	0	1	0	0	1	0	0	0	0	0	0	0	1	0	
Pinheiro, F. A. C. & Goguen, J. A. 1996	1	0	0	0	0	0	1	1	0	0	0	0	0	0	1	0	1	0	0	1	0	0	1	0	0	0	1	0	0	0	1	1	0	1	0	1	0	1	0	0	1	0	
Pohl, K. 1996	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	1	1	0	0	1	1	1	1	1	1	1	1	0	1	0	0	0	1	0	0	0	1		
Ramesh, B. & Jarke, M.	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	1	1	0	1	0	1	0	0	0	0	0	0	0	0	1	0		
Rose 2008	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	1	0	0	1	0	0	0	1	0	0	0	1	1	0	0	0	0	0	0	0	0	0	1		
Schwarz et.al 2009	0	0	1	0	0	0	0	0	0	0	0	0	0	1	1	1	0	1	0	0	0	1	0	0	1	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	1	
Sharif, B. & Maletic, J. I. 2007	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	1	0	1	0	1	0	1	0	0	0	1	0	1	0		
Sherba et.al 2003	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	0	1	0	0	0	1	1	0	1	0	0	0	0	0	0	0	0	0	0	1	0	
Sousa et.al 2008	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	1	0	0	1	0	1	1	1	0	1	0	0	0	1	0	0	1	0	0	1	0	
Spanoudakis et.al 2003	0	0	1	0	0	0	1	0	0	0	1	0	0	0	0	0	1	0	1	0	1	0	1	0	0	0	1	1	0	1	0	1	0	0	0	0	0	0	0	0	1	0	
Spanoudakis et.al 2004	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	1	0	1	0	1	0	1	0	1	0	0	0	1	1	0	1	0	1	0	0	0	0	0	0	0	1	0	
Spanoudakis, G. & Zisman, A. 2005	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	1	0	0	1	0	0	0	0	1	1	0	0	0	1	0	0	0	1	0	0	0	1	0		
Vanhooff et.al 2007b	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	1	0	1	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	1	0	1
von Knethen, A. & Grund, M. 2003	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	1	0	1	0	1	0	0	0	0	0	0	1	0	0	
Walderhaug et.al 2006	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	1	0	1	0	0	0	0	1	0	1	1	0	0	0	0	0	0	1	1	0
Wenzel et.al 2007	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	1	0	1	1	0	0	0	0	0	0	0	0	1	0	
Widen, J. B. T. 2001	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	1	0	0	0	0	0	1	0	1	1	0	0	0	1	0	0	1	0	0	1	0
Ying et.al 2004	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0
Zimmermann et.al 2004	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	
Zisman et.al 2003	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	1	0	1	0	0	0	0	0	1	0	0	1	0	1	0
Zou et.al 2006	0	1	0	1	0	1	0	0	0	0	0	1	0	0	0	1	0	0	0	1	0	0																					

Appendix C

Phenetic Analysis Results

In this appendix, the results of the phenetic analysis using the different similarity metrics are provided.

Table C.1: Resemblance Matrix using the Dice Coefficient - Part I

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	
0.00E+00	2.22E-01	3.33E-01	2.50E-01	4.12E-01	4.00E-01	3.75E-01	5.00E-01	3.75E-01	7.65E-01	6.84E-01	4.44E-01	5.00E-01	4.12E-01	6.67E-01	5.71E-01	2.94E-01	3.68E-01	3.33E-01	5.29E-01	5.29E-01	2.94E-01	2.94E-01	5.29E-01	4.44E-01	1
	0.00E+00	2.94E-01	2.22E-01	5.79E-01	4.55E-01	5.56E-01	5.56E-01	5.56E-01	7.89E-01	7.14E-01	4.00E-01	4.44E-01	5.79E-01	7.00E-01	6.25E-01	4.74E-01	5.24E-01	2.00E-01	4.74E-01	5.79E-01	4.74E-01	2.63E-01	4.74E-01	6.00E-01	2
		0.00E+00	3.33E-01	6.25E-01	5.79E-01	4.67E-01	6.00E-01	6.00E-01	6.25E-01	5.56E-01	2.94E-01	6.00E-01	5.00E-01	5.29E-01	5.38E-01	5.00E-01	5.56E-01	4.12E-01	6.25E-01	5.00E-01	5.00E-01	3.75E-01	6.25E-01	5.29E-01	3
			0.00E+00	6.47E-01	6.00E-01	6.25E-01	6.25E-01	6.25E-01	7.65E-01	6.84E-01	3.33E-01	3.75E-01	6.47E-01	6.67E-01	7.14E-01	5.29E-01	5.79E-01	1.11E-01	5.29E-01	6.47E-01	5.29E-01	2.94E-01	4.12E-01	6.67E-01	4
				0.00E+00	3.33E-01	1.76E-01	2.94E-01	1.76E-01	5.56E-01	7.00E-01	5.79E-01	5.29E-01	2.22E-01	6.84E-01	3.33E-01	2.22E-01	1.00E-01	6.84E-01	6.67E-01	7.78E-01	1.11E-01	5.56E-01	7.78E-01	1.58E-01	5
					0.00E+00	4.00E-01	4.00E-01	3.00E-01	5.24E-01	5.65E-01	4.55E-01	5.00E-01	4.29E-01	5.45E-01	4.44E-01	2.38E-01	3.04E-01	6.36E-01	6.19E-01	8.10E-01	2.38E-01	4.29E-01	7.14E-01	4.55E-01	6
						0.00E+00	3.75E-01	2.50E-01	5.29E-01	6.84E-01	5.56E-01	5.00E-01	5.88E-02	6.67E-01	4.29E-01	2.94E-01	2.63E-01	6.67E-01	6.47E-01	6.47E-01	1.76E-01	5.29E-01	7.65E-01	1.11E-01	7
							0.00E+00	1.25E-01	4.12E-01	5.79E-01	5.56E-01	5.00E-01	4.12E-01	5.56E-01	2.86E-01	2.94E-01	3.68E-01	6.67E-01	6.47E-01	6.47E-01	2.94E-01	5.29E-01	7.65E-01	4.44E-01	8
								0.00E+00	5.29E-01	6.84E-01	5.56E-01	5.00E-01	2.94E-01	6.67E-01	2.86E-01	1.76E-01	2.63E-01	6.67E-01	6.47E-01	7.65E-01	1.76E-01	5.29E-01	7.65E-01	3.33E-01	9
									0.00E+00	2.00E-01	4.74E-01	4.12E-01	5.56E-01	1.58E-01	4.67E-01	5.56E-01	6.00E-01	7.89E-01	8.89E-01	6.67E-01	5.56E-01	6.67E-01	7.78E-01	5.79E-01	10
										0.00E+00	4.29E-01	5.79E-01	7.00E-01	4.76E-02	6.47E-01	6.00E-01	6.36E-01	7.14E-01	8.00E-01	7.00E-01	6.00E-01	5.00E-01	6.00E-01	7.14E-01	11
											0.00E+00	4.44E-01	5.79E-01	4.00E-01	6.25E-01	4.74E-01	5.24E-01	4.00E-01	5.79E-01	6.84E-01	4.74E-01	4.74E-01	6.00E-01	6.00E-01	12
												0.00E+00	5.29E-01	5.56E-01	5.71E-01	5.29E-01	5.79E-01	4.44E-01	5.29E-01	7.65E-01	5.29E-01	2.94E-01	4.12E-01	5.56E-01	13
													0.00E+00	6.84E-01	4.67E-01	3.33E-01	2.00E-01	5.79E-01	6.67E-01	6.67E-01	2.22E-01	5.56E-01	7.78E-01	5.26E-02	14
														0.00E+00	6.25E-01	5.79E-01	6.19E-01	7.00E-01	7.89E-01	6.84E-01	5.79E-01	4.74E-01	5.79E-01	7.00E-01	15
															0.00E+00	2.00E-01	4.12E-01	7.50E-01	7.33E-01	3.33E-01	6.00E-01	8.67E-01	5.00E-01	5.00E-01	16
																0.00E+00	2.00E-01	5.79E-01	6.67E-01	7.78E-01	1.11E-01	4.44E-01	6.67E-01	3.68E-01	17
																	0.00E+00	5.24E-01	7.00E-01	8.00E-01	1.00E-01	5.00E-01	7.00E-01	1.43E-01	18
																		0.00E+00	5.79E-01	6.84E-01	5.79E-01	3.68E-01	4.74E-01	6.00E-01	19
																			0.00E+00	7.78E-01	6.67E-01	3.33E-01	2.22E-01	6.84E-01	20
																				0.00E+00	7.78E-01	6.67E-01	8.89E-01	6.84E-01	21
																					0.00E+00	4.44E-01	6.67E-01	2.63E-01	22
																						0.00E+00	3.33E-01	5.79E-01	23
																							0.00E+00	7.89E-01	24
																								0.00E+00	25

Table C.2: Resemblance Matrix using the Dice Coefficient - Part II

26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	
3.33E-01	5.00E-01	1.25E-01	4.00E-01	3.33E-01	5.00E-01	7.78E-01	5.56E-01	7.89E-01	6.25E-01	5.29E-01	4.12E-01	5.29E-01	2.94E-01	4.74E-01	6.47E-01	1.76E-01	5.79E-01	6.47E-01	5.00E-01	1
5.00E-01	6.67E-01	1.11E-01	5.45E-01	2.94E-01	4.55E-01	7.00E-01	5.00E-01	6.19E-01	6.67E-01	4.74E-01	3.68E-01	4.74E-01	4.74E-01	3.33E-01	5.79E-01	1.58E-01	4.29E-01	4.74E-01	4.44E-01	2
5.29E-01	4.67E-01	2.00E-01	5.79E-01	4.29E-01	5.79E-01	7.65E-01	5.29E-01	6.67E-01	7.33E-01	6.25E-01	5.00E-01	6.25E-01	5.00E-01	5.56E-01	5.00E-01	1.25E-01	6.67E-01	7.50E-01	6.00E-01	3
5.56E-01	7.50E-01	1.25E-01	6.00E-01	2.00E-01	5.00E-01	7.78E-01	4.44E-01	6.84E-01	6.25E-01	5.29E-01	2.94E-01	4.12E-01	5.29E-01	2.63E-01	5.29E-01	1.76E-01	3.68E-01	4.12E-01	5.00E-01	4
5.26E-02	6.47E-01	5.29E-01	1.43E-01	6.25E-01	6.19E-01	5.79E-01	6.84E-01	8.00E-01	8.82E-01	6.67E-01	6.67E-01	7.78E-01	2.22E-01	8.00E-01	6.67E-01	5.56E-01	8.00E-01	7.78E-01	6.47E-01	5
2.73E-01	7.00E-01	5.00E-01	2.50E-01	6.84E-01	5.83E-01	5.45E-01	4.55E-01	8.26E-01	9.00E-01	6.19E-01	6.19E-01	7.14E-01	2.38E-01	7.39E-01	6.19E-01	5.24E-01	7.39E-01	8.10E-01	7.00E-01	6
2.22E-01	5.00E-01	5.00E-01	3.00E-01	6.00E-01	6.00E-01	6.67E-01	6.67E-01	6.84E-01	8.75E-01	6.47E-01	6.47E-01	7.65E-01	2.94E-01	7.89E-01	5.29E-01	4.12E-01	7.89E-01	7.65E-01	6.25E-01	7
3.33E-01	7.50E-01	5.00E-01	4.00E-01	6.00E-01	6.00E-01	4.44E-01	6.67E-01	7.89E-01	7.50E-01	6.47E-01	6.47E-01	7.65E-01	1.76E-01	7.89E-01	6.47E-01	5.29E-01	7.89E-01	7.65E-01	6.25E-01	8
2.22E-01	6.25E-01	5.00E-01	3.00E-01	6.00E-01	6.00E-01	5.56E-01	6.67E-01	7.89E-01	8.75E-01	6.47E-01	6.47E-01	7.65E-01	5.88E-02	7.89E-01	6.47E-01	5.29E-01	7.89E-01	7.65E-01	6.25E-01	9
5.79E-01	7.65E-01	7.65E-01	6.19E-01	6.25E-01	8.10E-01	2.63E-01	3.68E-01	6.00E-01	6.47E-01	8.89E-01	6.67E-01	7.78E-01	5.56E-01	9.00E-01	5.56E-01	6.67E-01	9.00E-01	8.89E-01	8.82E-01	10
6.19E-01	7.89E-01	6.84E-01	6.52E-01	6.67E-01	8.26E-01	3.33E-01	4.29E-01	5.45E-01	6.84E-01	8.00E-01	6.00E-01	6.00E-01	8.18E-01	6.00E-01	6.00E-01	7.27E-01	8.00E-01	7.89E-01	7.89E-01	11
5.00E-01	6.67E-01	3.33E-01	4.55E-01	2.94E-01	6.36E-01	6.00E-01	5.00E-01	5.24E-01	6.67E-01	5.79E-01	3.68E-01	4.74E-01	4.74E-01	5.24E-01	3.68E-01	2.63E-01	6.19E-01	6.84E-01	6.67E-01	12
5.56E-01	8.75E-01	3.75E-01	6.00E-01	2.00E-01	5.00E-01	5.56E-01	2.22E-01	5.79E-01	6.25E-01	5.29E-01	2.94E-01	4.12E-01	5.29E-01	5.79E-01	4.12E-01	4.12E-01	5.79E-01	5.29E-01	5.00E-01	13
2.63E-01	5.29E-01	5.29E-01	2.38E-01	6.25E-01	6.19E-01	6.84E-01	6.84E-01	7.00E-01	8.82E-01	6.67E-01	6.67E-01	7.78E-01	3.33E-01	8.00E-01	5.56E-01	4.44E-01	8.00E-01	7.78E-01	6.47E-01	14
6.00E-01	7.78E-01	6.67E-01	6.36E-01	6.47E-01	8.18E-01	4.00E-01	4.00E-01	5.24E-01	6.67E-01	7.89E-01	4.74E-01	5.79E-01	5.79E-01	8.10E-01	5.79E-01	5.79E-01	7.14E-01	7.89E-01	7.78E-01	15
3.75E-01	7.14E-01	5.71E-01	4.44E-01	6.92E-01	6.67E-01	5.00E-01	6.25E-01	8.82E-01	8.57E-01	7.33E-01	6.00E-01	8.67E-01	3.33E-01	8.82E-01	7.33E-01	6.00E-01	8.82E-01	8.67E-01	7.14E-01	16
1.58E-01	6.47E-01	4.12E-01	2.38E-01	6.25E-01	5.24E-01	5.79E-01	5.79E-01	8.00E-01	8.82E-01	6.67E-01	4.44E-01	6.67E-01	1.11E-01	7.00E-01	5.56E-01	4.44E-01	7.00E-01	7.78E-01	6.47E-01	17
4.76E-02	6.84E-01	4.74E-01	4.35E-02	6.67E-01	5.65E-01	6.19E-01	6.19E-01	8.18E-01	8.95E-01	7.00E-01	6.00E-01	7.00E-01	2.00E-01	7.27E-01	6.00E-01	5.00E-01	7.27E-01	8.00E-01	6.84E-01	18
6.00E-01	7.78E-01	2.22E-01	5.45E-01	2.94E-01	5.45E-01	8.00E-01	5.00E-01	6.19E-01	6.67E-01	5.79E-01	3.68E-01	4.74E-01	5.79E-01	2.38E-01	5.79E-01	2.63E-01	3.33E-01	3.68E-01	5.56E-01	19
6.84E-01	8.82E-01	4.12E-01	6.19E-01	5.00E-01	4.29E-01	8.95E-01	6.84E-01	5.00E-01	8.82E-01	0.00E+00	4.44E-01	3.33E-01	6.67E-01	7.00E-01	5.56E-01	4.44E-01	3.00E-01	2.22E-01	1.76E-01	20
7.89E-01	2.94E-01	5.29E-01	8.10E-01	6.25E-01	6.19E-01	7.89E-01	7.89E-01	9.00E-01	1.76E-01	7.78E-01	7.78E-01	8.89E-01	7.78E-01	5.00E-01	8.89E-01	4.44E-01	9.00E-01	8.89E-01	7.65E-01	21
5.26E-02	6.47E-01	4.12E-01	1.43E-01	6.25E-01	5.24E-01	5.79E-01	5.79E-01	8.00E-01	8.82E-01	6.67E-01	5.56E-01	6.67E-01	1.11E-01	7.00E-01	5.56E-01	4.44E-01	7.00E-01	7.78E-01	6.47E-01	22
4.74E-01	7.65E-01	1.76E-01	5.24E-01	3.75E-01	4.29E-01	7.89E-01	3.68E-01	6.00E-01	7.65E-01	3.33E-01	2.22E-01	3.33E-01	4.44E-01	5.00E-01	5.56E-01	2.22E-01	4.00E-01	4.44E-01	2.94E-01	23
6.84E-01	1.00E+00	4.12E-01	7.14E-01	3.75E-01	4.29E-01	7.89E-01	4.74E-01	4.00E-01	7.65E-01	2.22E-01	2.22E-01	1.11E-01	6.67E-01	6.00E-01	3.33E-01	4.44E-01	2.00E-01	2.22E-01	2.94E-01	24
2.00E-01	5.56E-01	5.56E-01	1.82E-01	6.47E-01	6.36E-01	7.00E-01	7.00E-01	7.14E-01	8.89E-01	6.84E-01	6.84E-01	7.89E-01	3.68E-01	8.10E-01	5.79E-01	4.74E-01	8.10E-01	7.89E-01	6.67E-01	25
0.00E+00	6.67E-01	4.44E-01	9.09E-02	6.47E-01	5.45E-01	6.00E-01	6.00E-01	8.10E-01	8.89E-01	6.84E-01	5.79E-01	6.84E-01	1.58E-01	7.14E-01	5.79E-01	4.74E-01	7.14E-01	7.89E-01	6.67E-01	26
	0.00E+00	6.25E-01	7.00E-01	7.33E-01	6.00E-01	8.89E-01	8.89E-01	8.95E-01	5.00E-01	8.82E-01	8.82E-01	1.00E+00	6.47E-01	4.74E-01	8.82E-01	5.29E-01	1.00E+00	1.00E+00	8.75E-01	27
		0.00E+00	5.00E-01	2.00E-01	4.00E-01	7.78E-01	4.44E-01	6.84E-01	6.25E-01	4.12E-01	2.94E-01	4.12E-01	4.12E-01	3.68E-01	5.29E-01	5.88E-02	4.74E-01	5.29E-01	3.75E-01	28
			0.00E+00	6.84E-01	5.83E-01	6.36E-01	6.36E-01	8.26E-01	9.00E-01	6.19E-01	6.19E-01	7.14E-01	2.38E-01	7.39E-01	6.19E-01	5.24E-01	7.39E-01	8.10E-01	7.00E-01	29
				0.00E+00	5.79E-01	6.47E-01	4.12E-01	5.56E-01	4.67E-01	5.00E-01	2.50E-01	3.75E-01	6.25E-01	4.44E-01	5.00E-01	2.50E-01	5.56E-01	5.00E-01	4.67E-01	30
					0.00E+00	8.18E-01	5.45E-01	7.39E-01	7.00E-01	4.29E-01	5.24E-01	5.24E-01	5.24E-01	3.91E-01	5.24E-01	4.29E-01	4.78E-01	5.24E-01	5.00E-01	31
						0.00E+00	5.00E-01	7.14E-01	6.67E-01	8.95E-01	6.84E-01	7.89E-01	5.79E-01	9.05E-01	6.84E-01	7.89E-01	9.05E-01	8.95E-01	8.89E-01	32
							0.00E+00	6.19E-01	6.67E-01	6.84E-01	3.68E-01	4.74E-01	5.79E-01	6.19E-01	4.74E-01	4.74E-01	6.19E-01	6.84E-01	6.67E-01	33
								0.00E+00	8.95E-01	5.00E-01	8.00E-01	7.27E-01	3.00E-01	8.00E-01	7.27E-01	3.00E-01	3.64E-01	4.00E-01	3.68E-01	34
									0.00E+00	8.82E-01	6.47E-01	7.65E-01	8.82E-01	4.74E-01	8.82E-01	6.47E-01	8.95E-01	8.82E-01	8.75E-01	35
										0.00E+00	4.44E-01	3.33E-01	6.67E-01	7.00E-01	5.56E-01	4.44E-01	3.00E-01	2.22E-01	1.76E-01	36
											0.00E+00	2.22E-01	5.56E-01	5.00E-01	4.44E-01	3.33E-01	4.00E-01	4.44E-01	4.12E-01	37
												0.00E+00	6.67E-01	6.00E-01	2.22E-01	4.44E-01	2.00E-01	3.33E-01	1.76E-01	38
													0.00E+00	7.00E-01	5.56E-01	4.44E-01	7.00E-01	7.78E-01	6.47E-01	39
														0.00E+00	4.00E-01	4.55E-01	5.00E-01	6.84E-01	4.12E-01	40
															0.00E+00	4.44E-01	4.00E-01	5.56E-01	4.12E-01	41
																0.00E+00	5.00E-01	5.56E-01	4.12E-01	42
																	0.00E+00	1.00E-01	2.63E-01	43
																		0.00E+00	2.94E-01	44
																			0.00E+00	45

Table C.3: Resemblance Matrix using the Hamming Coefficient - Part I

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	
0.00E+00	1.32E-01	1.32E-01	1.05E-01	1.84E-01	2.11E-01	1.58E-01	2.11E-01	1.58E-01	3.42E-01	3.42E-01	2.11E-01	2.11E-01	1.84E-01	3.16E-01	2.11E-01	1.32E-01	1.84E-01	1.58E-01	2.37E-01	2.37E-01	1.32E-01	1.32E-01	2.37E-01	2.11E-01	1
	0.00E+00	2.11E-01	7.89E-02	3.16E-01	3.42E-01	2.89E-01	2.37E-01	2.89E-01	3.16E-01	3.16E-01	2.37E-01	2.37E-01	3.16E-01	2.89E-01	2.89E-01	2.63E-01	3.16E-01	7.89E-02	3.16E-01	2.63E-01	2.63E-01	2.11E-01	2.63E-01	3.42E-01	2
		0.00E+00	1.32E-01	2.63E-01	2.89E-01	1.84E-01	2.37E-01	2.37E-01	2.63E-01	2.63E-01	1.32E-01	2.37E-01	2.11E-01	2.37E-01	1.84E-01	2.11E-01	2.63E-01	1.84E-01	2.63E-01	2.11E-01	2.11E-01	1.58E-01	2.63E-01	2.37E-01	3
			0.00E+00	2.89E-01	3.16E-01	2.63E-01	2.63E-01	3.42E-01	3.42E-01	1.58E-01	1.58E-01	2.89E-01	3.16E-01	2.63E-01	2.37E-01	2.89E-01	5.26E-02	2.37E-01	2.89E-01	2.37E-01	2.37E-01	1.32E-01	1.84E-01	3.16E-01	4
				0.00E+00	1.84E-01	7.89E-02	1.32E-01	7.89E-02	2.63E-01	3.68E-01	2.89E-01	2.37E-01	1.05E-01	3.42E-01	1.32E-01	1.05E-01	5.26E-02	3.42E-01	3.16E-01	3.68E-01	5.26E-02	2.63E-01	3.68E-01	7.89E-02	5
					0.00E+00	2.11E-01	2.11E-01	1.58E-01	2.89E-01	3.42E-01	2.63E-01	2.63E-01	2.37E-01	3.16E-01	2.11E-01	1.32E-01	1.84E-01	3.68E-01	3.42E-01	4.47E-01	1.32E-01	2.37E-01	3.95E-01	2.63E-01	6
						0.00E+00	1.58E-01	1.05E-01	2.37E-01	3.42E-01	2.63E-01	2.11E-01	2.63E-02	3.16E-01	1.58E-01	1.32E-01	1.32E-01	3.16E-01	2.89E-01	2.89E-01	7.89E-02	2.37E-01	3.42E-01	5.26E-02	7
							0.00E+00	5.26E-02	1.84E-01	2.89E-01	2.63E-01	2.11E-01	1.84E-01	2.63E-01	1.05E-01	1.32E-01	1.84E-01	3.16E-01	2.89E-01	2.89E-01	1.32E-01	2.37E-01	3.42E-01	2.11E-01	8
								0.00E+00	2.37E-01	3.42E-01	2.63E-01	2.11E-01	1.32E-01	3.16E-01	1.05E-01	7.89E-02	1.32E-01	3.16E-01	2.89E-01	3.42E-01	7.89E-02	2.37E-01	3.42E-01	1.58E-01	9
									0.00E+00	1.05E-01	2.37E-01	1.84E-01	2.63E-01	7.89E-02	1.84E-01	2.63E-01	3.16E-01	3.95E-01	4.21E-01	3.16E-01	2.63E-01	3.16E-01	3.68E-01	2.89E-01	10
										0.00E+00	2.37E-01	2.89E-01	3.68E-01	2.63E-02	2.89E-01	3.16E-01	3.68E-01	3.95E-01	4.21E-01	3.68E-01	3.16E-01	2.63E-01	3.16E-01	3.95E-01	11
											0.00E+00	2.11E-01	2.89E-01	2.11E-01	2.63E-01	2.37E-01	2.89E-01	2.11E-01	2.89E-01	3.42E-01	2.37E-01	2.37E-01	2.37E-01	3.16E-01	12
												0.00E+00	2.37E-01	2.63E-01	2.11E-01	2.37E-01	2.89E-01	2.11E-01	2.37E-01	3.42E-01	2.37E-01	1.32E-01	1.84E-01	2.63E-01	13
													0.00E+00	3.42E-01	1.84E-01	1.58E-01	1.05E-01	2.89E-01	3.16E-01	3.16E-01	1.05E-01	2.63E-01	3.68E-01	2.63E-02	14
														0.00E+00	2.63E-01	2.89E-01	3.42E-01	3.68E-01	3.95E-01	3.42E-01	2.89E-01	2.37E-01	2.89E-01	3.68E-01	15
															0.00E+00	7.89E-02	1.84E-01	3.16E-01	2.89E-01	2.89E-01	1.32E-01	2.37E-01	3.42E-01	2.11E-01	16
																0.00E+00	1.05E-01	2.89E-01	3.16E-01	3.68E-01	5.26E-02	2.11E-01	3.16E-01	1.84E-01	17
																	0.00E+00	2.89E-01	3.68E-01	4.21E-01	5.26E-02	2.63E-01	3.68E-01	7.89E-02	18
																		0.00E+00	2.89E-01	3.42E-01	2.89E-01	1.84E-01	2.37E-01	3.16E-01	19
																			0.00E+00	3.68E-01	3.16E-01	1.58E-01	1.05E-01	3.42E-01	20
																				0.00E+00	3.68E-01	3.16E-01	4.21E-01	3.42E-01	21
																					0.00E+00	2.11E-01	3.16E-01	1.32E-01	22
																						0.00E+00	1.58E-01	2.89E-01	23
																							0.00E+00	3.95E-01	24
																								0.00E+00	25

Table C.4: Resemblance Matrix using the Hamming Coefficient - Part II

26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	
1.58E-01	2.11E-01	5.26E-02	2.11E-01	1.32E-01	2.63E-01	3.68E-01	2.63E-01	3.95E-01	2.63E-01	2.37E-01	1.84E-01	2.37E-01	1.32E-01	2.37E-01	2.89E-01	7.89E-02	2.89E-01	2.89E-01	2.11E-01	1
2.89E-01	3.42E-01	1.32E-01	3.42E-01	1.58E-01	3.42E-01	3.42E-01	2.89E-01	3.68E-01	2.37E-01	3.16E-01	2.11E-01	2.63E-01	2.63E-01	1.58E-01	3.16E-01	1.58E-01	2.11E-01	2.11E-01	2.89E-01	2
2.37E-01	1.84E-01	7.89E-02	2.89E-01	1.58E-01	2.89E-01	3.42E-01	2.37E-01	3.16E-01	2.89E-01	2.63E-01	2.11E-01	2.63E-01	2.11E-01	2.63E-01	2.11E-01	5.26E-02	3.16E-01	3.16E-01	2.37E-01	3
2.63E-01	3.16E-01	5.26E-02	3.16E-01	7.89E-02	2.63E-01	3.68E-01	2.11E-01	3.42E-01	2.63E-01	2.37E-01	1.84E-01	1.32E-01	1.84E-01	2.37E-01	1.32E-01	7.89E-02	1.84E-01	1.84E-01	2.11E-01	4
2.63E-02	2.89E-01	2.37E-01	7.89E-02	2.63E-01	3.42E-01	2.89E-01	3.42E-01	4.21E-01	3.95E-01	3.16E-01	3.16E-01	3.68E-01	1.05E-01	4.21E-01	3.16E-01	2.63E-01	4.21E-01	3.68E-01	2.89E-01	5
1.58E-01	3.68E-01	2.63E-01	1.58E-01	3.42E-01	3.68E-01	3.16E-01	2.63E-01	5.00E-01	4.74E-01	3.42E-01	3.42E-01	3.95E-01	1.32E-01	4.47E-01	3.42E-01	2.89E-01	4.47E-01	4.47E-01	3.68E-01	6
1.05E-01	2.11E-01	2.11E-01	1.58E-01	2.37E-01	3.16E-01	3.16E-01	3.16E-01	3.42E-01	3.68E-01	2.89E-01	2.89E-01	3.42E-01	1.32E-01	3.95E-01	2.37E-01	1.84E-01	3.95E-01	3.42E-01	2.63E-01	7
1.58E-01	3.16E-01	2.11E-01	2.11E-01	2.37E-01	3.16E-01	2.11E-01	3.16E-01	3.95E-01	3.16E-01	2.89E-01	2.89E-01	3.42E-01	7.89E-02	3.95E-01	2.89E-01	2.37E-01	3.95E-01	3.42E-01	2.63E-01	8
1.05E-01	2.63E-01	2.11E-01	1.58E-01	2.37E-01	3.16E-01	2.63E-01	3.16E-01	3.95E-01	3.68E-01	2.89E-01	2.89E-01	3.42E-01	2.63E-02	3.95E-01	2.89E-01	2.37E-01	3.95E-01	3.42E-01	2.63E-01	9
2.89E-01	3.42E-01	3.42E-01	3.42E-01	2.63E-01	4.47E-01	1.32E-01	1.84E-01	3.16E-01	2.89E-01	4.21E-01	3.16E-01	3.68E-01	2.63E-01	4.74E-01	2.63E-01	3.16E-01	4.74E-01	4.21E-01	3.95E-01	10
3.42E-01	3.95E-01	3.42E-01	3.95E-01	3.16E-01	5.00E-01	1.84E-01	2.37E-01	3.16E-01	3.42E-01	4.21E-01	3.16E-01	2.63E-01	3.16E-01	4.74E-01	3.16E-01	4.21E-01	4.21E-01	4.21E-01	3.95E-01	11
2.63E-01	3.16E-01	1.58E-01	2.63E-01	1.32E-01	3.68E-01	3.16E-01	2.63E-01	2.89E-01	3.16E-01	2.89E-01	1.84E-01	2.37E-01	2.37E-01	2.89E-01	1.84E-01	1.32E-01	3.42E-01	3.42E-01	3.16E-01	12
2.63E-01	3.68E-01	1.58E-01	3.16E-01	7.89E-02	2.63E-01	2.63E-01	1.05E-01	2.89E-01	2.63E-01	2.37E-01	1.32E-01	1.84E-01	2.37E-01	2.89E-01	1.84E-01	1.84E-01	2.89E-01	2.37E-01	2.11E-01	13
1.32E-01	2.37E-01	2.37E-01	1.32E-01	2.63E-01	3.42E-01	3.42E-01	3.42E-01	3.68E-01	3.95E-01	3.16E-01	3.16E-01	3.68E-01	1.58E-01	4.21E-01	2.11E-01	4.21E-01	2.11E-01	3.68E-01	2.89E-01	14
3.16E-01	3.68E-01	3.16E-01	3.68E-01	2.89E-01	4.74E-01	2.11E-01	2.11E-01	2.89E-01	3.16E-01	3.95E-01	2.37E-01	2.89E-01	2.89E-01	4.47E-01	2.89E-01	2.89E-01	3.95E-01	3.95E-01	3.68E-01	15
1.58E-01	2.63E-01	2.11E-01	2.11E-01	2.37E-01	3.16E-01	2.11E-01	2.63E-01	3.95E-01	3.16E-01	2.89E-01	2.37E-01	3.42E-01	1.32E-01	3.95E-01	2.89E-01	2.37E-01	3.95E-01	3.42E-01	2.63E-01	16
7.89E-02	2.89E-01	1.84E-01	1.32E-01	2.63E-01	2.89E-01	2.89E-01	2.89E-01	4.21E-01	3.95E-01	3.16E-01	2.11E-01	3.16E-01	5.26E-02	3.68E-01	2.63E-01	2.11E-01	3.68E-01	3.68E-01	2.89E-01	17
2.63E-02	3.42E-01	2.37E-01	2.63E-02	3.16E-01	3.42E-01	3.42E-01	3.42E-01	4.74E-01	4.47E-01	3.68E-01	3.16E-01	3.68E-01	1.05E-01	4.21E-01	3.16E-01	2.63E-01	4.21E-01	4.21E-01	3.42E-01	18
3.16E-01	3.68E-01	1.05E-01	3.16E-01	1.32E-01	3.16E-01	4.21E-01	2.63E-01	3.42E-01	3.16E-01	2.89E-01	1.84E-01	2.37E-01	2.89E-01	1.32E-01	2.89E-01	1.32E-01	1.84E-01	1.84E-01	2.63E-01	19
3.42E-01	3.95E-01	1.84E-01	3.42E-01	2.11E-01	2.37E-01	4.47E-01	3.42E-01	2.63E-01	3.95E-01	0.00E+00	2.11E-01	1.58E-01	3.16E-01	3.68E-01	2.63E-01	2.11E-01	1.58E-01	1.05E-01	7.89E-02	20
3.95E-01	1.32E-01	2.37E-01	4.47E-01	2.63E-01	3.42E-01	3.95E-01	3.95E-01	4.74E-01	7.89E-02	3.68E-01	3.68E-01	4.21E-01	3.68E-01	2.63E-01	4.21E-01	2.11E-01	4.74E-01	4.21E-01	3.42E-01	21
2.63E-02	2.89E-01	1.84E-01	7.89E-02	2.63E-01	2.89E-01	2.89E-01	2.89E-01	4.21E-01	3.95E-01	3.16E-01	2.63E-01	3.16E-01	5.26E-02	3.68E-01	2.63E-01	2.11E-01	3.68E-01	3.68E-01	2.89E-01	22
2.37E-01	3.42E-01	7.89E-02	2.89E-01	1.58E-01	2.37E-01	3.95E-01	1.84E-01	3.16E-01	3.42E-01	1.58E-01	1.05E-01	1.58E-01	2.11E-01	2.63E-01	2.63E-01	1.05E-01	2.11E-01	2.11E-01	1.32E-01	23
3.42E-01	4.47E-01	1.84E-01	3.95E-01	1.58E-01	2.37E-01	3.95E-01	2.37E-01	2.11E-01	3.42E-01	1.05E-01	1.05E-01	5.26E-02	3.16E-01	3.16E-01	1.58E-01	2.11E-01	1.05E-01	1.05E-01	1.32E-01	24
1.05E-01	2.63E-01	2.63E-01	1.05E-01	2.89E-01	3.68E-01	3.68E-01	3.68E-01	3.95E-01	4.21E-01	3.42E-01	3.42E-01	3.95E-01	1.84E-01	4.47E-01	2.89E-01	2.37E-01	4.47E-01	3.95E-01	3.16E-01	25
0.00E+00	3.16E-01	2.11E-01	5.26E-02	2.89E-01	3.16E-01	3.16E-01	3.16E-01	4.47E-01	4.21E-01	3.42E-01	2.89E-01	3.42E-01	7.89E-02	3.95E-01	2.89E-01	2.37E-01	3.95E-01	3.95E-01	3.16E-01	26
	0.00E+00			0.00E+00	3.68E-01	2.89E-01	3.16E-01	4.21E-01	4.47E-01	2.11E-01	3.95E-01	3.95E-01	4.47E-01	2.89E-01	2.37E-01	3.95E-01	5.00E-01	4.47E-01	3.68E-01	27
		0.00E+00	2.63E-01	7.89E-02	2.11E-01	3.68E-01	2.11E-01	3.42E-01	2.63E-01	1.84E-01	1.32E-01	1.84E-01	1.84E-01	1.84E-01	2.37E-01	2.63E-02	2.37E-01	2.37E-01	1.58E-01	28
			0.00E+00		3.68E-01	3.68E-01	3.68E-01	5.00E-01	4.74E-01	3.42E-01	3.42E-01	3.95E-01	1.32E-01	4.47E-01	3.42E-01	2.89E-01	4.47E-01	4.47E-01	3.68E-01	29
				0.00E+00	2.89E-01	2.89E-01	1.84E-01	2.63E-01	1.84E-01	2.11E-01	1.05E-01	1.58E-01	2.63E-01	2.11E-01	2.11E-01	1.05E-01	2.63E-01	2.11E-01	1.84E-01	30
					0.00E+00	4.74E-01	3.16E-01	4.47E-01	3.68E-01	2.37E-01	2.89E-01	2.89E-01	2.37E-01	2.89E-01	2.37E-01	2.89E-01	2.37E-01	2.89E-01	2.63E-01	31
						0.00E+00	2.63E-01	3.95E-01	3.16E-01	4.47E-01	3.42E-01	3.95E-01	2.89E-01	5.00E-01	3.42E-01	3.95E-01	5.00E-01	4.47E-01	4.21E-01	32
							0.00E+00	3.42E-01	3.16E-01	3.42E-01	1.84E-01	2.37E-01	2.89E-01	3.42E-01	2.37E-01	2.37E-01	3.42E-01	3.42E-01	3.16E-01	33
								0.00E+00	4.47E-01	2.63E-01	2.63E-01	1.58E-01	4.21E-01	1.58E-01	3.16E-01	2.11E-01	2.11E-01	2.11E-01	1.84E-01	34
									0.00E+00	3.95E-01	2.89E-01	3.42E-01	3.95E-01	2.37E-01	3.95E-01	2.89E-01	4.47E-01	3.95E-01	3.68E-01	35
										0.00E+00	2.11E-01	1.58E-01	3.16E-01	3.68E-01	2.63E-01	2.11E-01	1.58E-01	1.05E-01	7.89E-02	36
											0.00E+00	1.05E-01	2.63E-01	2.63E-01	2.11E-01	1.58E-01	2.11E-01	2.11E-01	1.84E-01	37
												0.00E+00	3.16E-01	3.16E-01	1.05E-01	2.11E-01	1.05E-01	1.58E-01	7.89E-02	38
													0.00E+00	3.68E-01	2.63E-01	2.11E-01	3.68E-01	3.68E-01	2.89E-01	39
														0.00E+00	3.68E-01	2.11E-01	2.63E-01	2.63E-01	3.42E-01	40
															0.00E+00	2.11E-01	2.11E-01	2.63E-01	1.84E-01	41
																0.00E+00	2.63E-01	2.63E-01	1.84E-01	42
																	0.00E+00	5.26E-02	1.32E-01	43
																		0.00E+00	1.32E-01	44
																			0.00E+00	45

Table C.5: Resemblance Matrix using the Jaccard Coefficient - Part I

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	
0.00E+00	4.55E-01	5.00E-01	4.00E-01	5.83E-01	5.71E-01	5.45E-01	6.67E-01	5.45E-01	8.67E-01	8.13E-01	6.15E-01	6.67E-01	5.83E-01	8.00E-01	7.27E-01	4.55E-01	5.38E-01	5.00E-01	6.92E-01	6.92E-01	4.55E-01	4.55E-01	6.92E-01	6.15E-01	1
	0.00E+00	6.67E-01	3.00E-01	8.00E-01	7.65E-01	7.86E-01	6.92E-01	7.86E-01	8.00E-01	7.50E-01	6.43E-01	6.92E-01	8.00E-01	7.33E-01	8.46E-01	7.14E-01	7.50E-01	2.73E-01	8.00E-01	7.14E-01	7.14E-01	6.15E-01	7.14E-01	8.13E-01	2
		0.00E+00	5.00E-01	7.69E-01	7.33E-01	6.36E-01	7.50E-01	7.50E-01	7.69E-01	7.14E-01	4.55E-01	7.50E-01	6.67E-01	6.92E-01	7.00E-01	6.67E-01	7.14E-01	5.83E-01	7.69E-01	6.67E-01	6.67E-01	5.45E-01	7.69E-01	6.92E-01	3
			0.00E+00	7.86E-01	7.50E-01	7.69E-01	7.69E-01	7.69E-01	8.67E-01	8.13E-01	5.00E-01	5.45E-01	7.86E-01	8.00E-01	8.33E-01	6.92E-01	7.33E-01	2.00E-01	6.92E-01	7.86E-01	6.92E-01	4.55E-01	5.83E-01	8.00E-01	4
				0.00E+00	5.00E-01	3.00E-01	4.55E-01	3.00E-01	7.14E-01	8.24E-01	7.33E-01	6.92E-01	3.64E-01	8.13E-01	5.00E-01	3.64E-01	1.82E-01	8.13E-01	8.00E-01	8.75E-01	2.00E-01	7.14E-01	8.75E-01	2.73E-01	5
					0.00E+00	5.71E-01	5.71E-01	4.62E-01	6.88E-01	7.22E-01	6.25E-01	6.67E-01	6.00E-01	7.06E-01	6.15E-01	3.85E-01	4.67E-01	7.78E-01	7.65E-01	8.95E-01	3.85E-01	6.00E-01	8.33E-01	6.25E-01	6
						0.00E+00	5.45E-01	4.00E-01	6.92E-01	8.13E-01	7.14E-01	6.67E-01	1.11E-01	8.00E-01	6.00E-01	4.55E-01	4.17E-01	8.00E-01	7.86E-01	7.86E-01	3.00E-01	6.92E-01	8.67E-01	2.00E-01	7
							0.00E+00	2.22E-01	5.83E-01	7.33E-01	7.14E-01	6.67E-01	5.83E-01	7.14E-01	4.44E-01	4.55E-01	5.38E-01	8.00E-01	7.86E-01	7.86E-01	4.55E-01	6.92E-01	8.67E-01	6.15E-01	8
								0.00E+00	6.92E-01	8.13E-01	7.14E-01	6.67E-01	4.55E-01	8.00E-01	4.44E-01	3.00E-01	4.17E-01	8.00E-01	7.86E-01	8.67E-01	3.00E-01	6.92E-01	8.67E-01	5.00E-01	9
									0.00E+00	3.33E-01	6.43E-01	5.83E-01	7.14E-01	2.73E-01	6.36E-01	7.14E-01	7.50E-01	8.82E-01	9.41E-01	8.00E-01	7.14E-01	8.00E-01	8.75E-01	7.33E-01	10
										0.00E+00	6.00E-01	7.33E-01	8.24E-01	9.09E-02	7.86E-01	7.50E-01	7.78E-01	8.33E-01	8.89E-01	8.24E-01	7.50E-01	6.67E-01	7.50E-01	8.33E-01	11
											0.00E+00	6.15E-01	7.33E-01	5.71E-01	7.69E-01	6.43E-01	6.88E-01	5.71E-01	7.33E-01	8.13E-01	6.43E-01	6.43E-01	6.43E-01	7.50E-01	12
												0.00E+00	6.92E-01	7.14E-01	7.27E-01	6.92E-01	7.33E-01	6.15E-01	6.92E-01	8.67E-01	6.92E-01	4.55E-01	5.83E-01	7.14E-01	13
													0.00E+00	8.13E-01	6.36E-01	5.00E-01	3.33E-01	7.33E-01	8.00E-01	8.00E-01	3.64E-01	7.14E-01	8.75E-01	1.00E-01	14
														0.00E+00	7.69E-01	7.33E-01	7.65E-01	8.24E-01	8.82E-01	8.13E-01	7.33E-01	6.43E-01	7.33E-01	8.24E-01	15
															0.00E+00	3.33E-01	5.83E-01	8.57E-01	8.46E-01	8.46E-01	5.00E-01	7.50E-01	9.29E-01	6.67E-01	16
																0.00E+00	3.33E-01	7.33E-01	8.00E-01	8.75E-01	2.00E-01	6.15E-01	8.00E-01	5.38E-01	17
																	0.00E+00	6.88E-01	8.24E-01	8.89E-01	1.82E-01	6.67E-01	8.24E-01	2.50E-01	18
																		0.00E+00	7.33E-01	8.13E-01	7.33E-01	5.38E-01	6.43E-01	7.50E-01	19
																			0.00E+00	8.75E-01	8.00E-01	5.00E-01	3.64E-01	8.13E-01	20
																				0.00E+00	8.75E-01	8.00E-01	9.41E-01	8.13E-01	21
																					0.00E+00	6.15E-01	8.00E-01	4.17E-01	22
																						0.00E+00	5.00E-01	7.33E-01	23
																							0.00E+00	8.82E-01	24
																								0.00E+00	25

Table C.6: Resemblance Matrix using the Jaccard Coefficient - Part II

26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	
5.00E-01	6.67E-01	2.22E-01	5.71E-01	5.00E-01	6.67E-01	8.75E-01	7.14E-01	8.82E-01	7.69E-01	6.92E-01	5.83E-01	6.92E-01	4.55E-01	6.43E-01	7.86E-01	3.00E-01	7.33E-01	7.86E-01	6.67E-01	1
7.33E-01	8.67E-01	4.55E-01	7.65E-01	5.45E-01	7.65E-01	8.13E-01	7.33E-01	8.24E-01	6.92E-01	8.00E-01	6.15E-01	7.14E-01	7.14E-01	4.62E-01	8.00E-01	5.00E-01	5.71E-01	6.15E-01	7.86E-01	2
6.92E-01	6.36E-01	3.33E-01	7.33E-01	6.00E-01	7.33E-01	8.67E-01	6.92E-01	8.00E-01	8.46E-01	7.69E-01	6.67E-01	7.69E-01	6.67E-01	7.14E-01	6.67E-01	2.22E-01	8.00E-01	8.57E-01	7.50E-01	3
7.14E-01	8.57E-01	2.22E-01	7.50E-01	3.33E-01	6.67E-01	8.75E-01	6.15E-01	8.13E-01	7.69E-01	6.92E-01	4.55E-01	5.83E-01	6.92E-01	4.17E-01	6.92E-01	3.00E-01	5.38E-01	5.83E-01	6.67E-01	4
1.00E-01	7.86E-01	6.92E-01	2.50E-01	7.69E-01	7.65E-01	7.33E-01	8.13E-01	8.89E-01	9.38E-01	8.00E-01	8.00E-01	8.75E-01	3.64E-01	8.89E-01	8.00E-01	7.14E-01	8.89E-01	8.75E-01	7.86E-01	5
4.29E-01	8.24E-01	6.67E-01	4.00E-01	8.13E-01	7.37E-01	7.06E-01	6.25E-01	9.05E-01	9.47E-01	7.65E-01	7.65E-01	8.33E-01	3.85E-01	8.50E-01	7.65E-01	6.88E-01	8.50E-01	8.95E-01	8.24E-01	6
3.64E-01	6.67E-01	6.67E-01	4.62E-01	7.50E-01	7.50E-01	8.00E-01	8.00E-01	8.13E-01	9.33E-01	7.86E-01	7.86E-01	8.67E-01	4.55E-01	8.82E-01	6.92E-01	5.83E-01	8.82E-01	8.67E-01	7.69E-01	7
5.00E-01	8.57E-01	6.67E-01	5.71E-01	7.50E-01	7.50E-01	6.15E-01	8.00E-01	8.82E-01	8.57E-01	7.86E-01	7.86E-01	8.67E-01	3.00E-01	8.82E-01	7.86E-01	6.92E-01	8.82E-01	8.67E-01	7.69E-01	8
3.64E-01	7.69E-01	6.67E-01	4.62E-01	7.50E-01	7.50E-01	7.14E-01	8.00E-01	8.82E-01	9.33E-01	7.86E-01	7.86E-01	8.67E-01	1.11E-01	8.82E-01	7.86E-01	6.92E-01	8.82E-01	8.67E-01	7.69E-01	9
7.33E-01	8.67E-01	8.67E-01	7.65E-01	7.69E-01	8.95E-01	4.17E-01	5.38E-01	7.50E-01	7.86E-01	9.41E-01	8.00E-01	8.75E-01	7.14E-01	9.47E-01	7.14E-01	8.00E-01	9.47E-01	9.41E-01	9.38E-01	10
7.65E-01	8.82E-01	8.13E-01	7.89E-01	8.00E-01	9.05E-01	5.00E-01	6.00E-01	7.06E-01	8.13E-01	8.89E-01	7.50E-01	7.50E-01	9.00E-01	9.00E-01	7.50E-01	8.42E-01	8.89E-01	8.82E-01	8.82E-01	11
6.67E-01	8.00E-01	5.00E-01	6.25E-01	4.55E-01	7.78E-01	7.50E-01	6.67E-01	6.88E-01	8.00E-01	7.33E-01	5.38E-01	6.43E-01	6.43E-01	6.88E-01	5.38E-01	4.17E-01	7.65E-01	8.13E-01	8.00E-01	12
7.14E-01	9.33E-01	5.45E-01	7.50E-01	3.33E-01	6.67E-01	7.14E-01	3.64E-01	7.33E-01	7.69E-01	6.92E-01	4.55E-01	5.83E-01	6.92E-01	7.33E-01	5.83E-01	5.83E-01	7.33E-01	6.92E-01	6.67E-01	13
4.17E-01	6.92E-01	6.92E-01	3.85E-01	7.69E-01	7.65E-01	8.13E-01	8.13E-01	8.24E-01	9.38E-01	8.00E-01	8.00E-01	8.75E-01	5.00E-01	8.89E-01	7.14E-01	6.15E-01	8.89E-01	8.75E-01	7.86E-01	14
7.50E-01	8.75E-01	8.00E-01	7.78E-01	7.86E-01	9.00E-01	5.71E-01	5.71E-01	6.88E-01	8.00E-01	8.82E-01	6.43E-01	7.33E-01	7.33E-01	8.95E-01	7.33E-01	7.33E-01	8.33E-01	8.82E-01	8.75E-01	15
5.45E-01	8.33E-01	7.27E-01	6.15E-01	8.18E-01	8.00E-01	6.67E-01	7.69E-01	9.38E-01	9.23E-01	8.46E-01	7.50E-01	9.29E-01	5.00E-01	9.38E-01	8.46E-01	7.50E-01	9.38E-01	9.29E-01	8.33E-01	16
2.73E-01	7.86E-01	5.83E-01	3.85E-01	7.69E-01	6.88E-01	7.33E-01	7.33E-01	8.89E-01	9.38E-01	8.00E-01	6.15E-01	8.00E-01	2.00E-01	8.24E-01	7.14E-01	6.15E-01	8.24E-01	8.75E-01	7.86E-01	17
9.09E-02	8.13E-01	6.43E-01	8.33E-02	8.00E-01	7.22E-01	7.65E-01	7.65E-01	9.00E-01	9.44E-01	8.24E-01	7.50E-01	8.24E-01	3.33E-01	8.42E-01	7.50E-01	6.67E-01	8.42E-01	8.89E-01	8.13E-01	18
7.50E-01	8.75E-01	3.64E-01	7.06E-01	4.55E-01	7.06E-01	8.89E-01	6.67E-01	7.65E-01	8.00E-01	7.33E-01	5.38E-01	6.43E-01	7.33E-01	3.85E-01	7.33E-01	4.17E-01	5.00E-01	5.38E-01	7.14E-01	19
8.13E-01	9.38E-01	5.83E-01	7.65E-01	6.67E-01	6.00E-01	9.44E-01	8.13E-01	6.67E-01	9.38E-01	0.00E+00	6.15E-01	5.00E-01	8.00E-01	8.24E-01	7.14E-01	6.15E-01	4.62E-01	3.64E-01	3.00E-01	20
8.82E-01	4.55E-01	6.92E-01	8.95E-01	7.69E-01	7.65E-01	8.82E-01	8.82E-01	9.47E-01	3.00E-01	8.75E-01	8.75E-01	9.41E-01	8.75E-01	6.67E-01	9.41E-01	6.15E-01	9.47E-01	9.41E-01	8.67E-01	21
1.00E-01	7.86E-01	5.83E-01	2.50E-01	7.69E-01	6.88E-01	7.33E-01	7.33E-01	8.89E-01	9.38E-01	8.00E-01	7.14E-01	8.00E-01	2.00E-01	8.24E-01	7.14E-01	6.15E-01	8.24E-01	8.75E-01	7.86E-01	22
6.43E-01	8.67E-01	3.00E-01	6.88E-01	5.45E-01	6.00E-01	8.82E-01	5.38E-01	7.50E-01	8.67E-01	5.00E-01	3.64E-01	5.00E-01	6.15E-01	6.67E-01	7.14E-01	3.64E-01	5.71E-01	6.15E-01	4.55E-01	23
8.13E-01	1.00E+00	5.83E-01	8.33E-01	5.45E-01	6.00E-01	8.82E-01	6.43E-01	5.71E-01	8.67E-01	3.64E-01	3.64E-01	2.00E-01	8.00E-01	7.50E-01	5.00E-01	6.15E-01	3.33E-01	3.64E-01	4.55E-01	24
3.33E-01	7.14E-01	7.14E-01	3.08E-01	7.86E-01	7.78E-01	8.24E-01	8.24E-01	8.33E-01	9.41E-01	8.13E-01	8.13E-01	8.82E-01	5.38E-01	8.95E-01	7.33E-01	6.43E-01	8.95E-01	8.82E-01	8.00E-01	25
0.00E+00	8.00E-01	6.15E-01	1.67E-01	7.86E-01	7.06E-01	7.50E-01	7.50E-01	8.95E-01	9.41E-01	8.13E-01	7.33E-01	8.13E-01	2.73E-01	8.33E-01	7.33E-01	6.43E-01	8.33E-01	8.82E-01	8.00E-01	26
	0.00E+00	7.69E-01	8.24E-01	8.46E-01	7.50E-01	9.41E-01	9.41E-01	9.44E-01	6.67E-01	9.38E-01	9.38E-01	1.00E+00	7.86E-01	6.43E-01	9.38E-01	6.92E-01	1.00E+00	1.00E+00	9.33E-01	27
		0.00E+00	6.67E-01	3.33E-01	5.71E-01	8.75E-01	6.15E-01	8.13E-01	7.69E-01	5.83E-01	4.55E-01	5.83E-01	5.83E-01	5.38E-01	6.92E-01	1.11E-01	6.43E-01	6.92E-01	5.45E-01	28
			0.00E+00	7.33E-01	7.86E-01	7.86E-01	7.86E-01	7.86E-01	9.05E-01	9.47E-01	7.65E-01	7.65E-01	8.33E-01	3.85E-01	8.50E-01	7.65E-01	6.88E-01	8.50E-01	8.95E-01	29
				0.00E+00	7.33E-01	7.86E-01	5.83E-01	7.14E-01	6.36E-01	6.67E-01	4.00E-01	5.45E-01	7.69E-01	6.15E-01	6.67E-01	4.00E-01	7.14E-01	6.67E-01	6.36E-01	30
					0.00E+00	9.00E-01	7.06E-01	8.50E-01	8.24E-01	6.00E-01	6.88E-01	6.88E-01	6.88E-01	5.63E-01	6.88E-01	6.00E-01	6.47E-01	6.88E-01	6.67E-01	31
						0.00E+00	6.67E-01	8.33E-01	8.00E-01	9.44E-01	8.13E-01	8.82E-01	7.33E-01	9.50E-01	8.13E-01	8.82E-01	9.50E-01	9.44E-01	9.41E-01	32
							0.00E+00	7.65E-01	8.00E-01	8.13E-01	5.38E-01	6.43E-01	7.33E-01	7.65E-01	6.43E-01	6.43E-01	7.65E-01	8.13E-01	8.00E-01	33
								0.00E+00	9.44E-01	6.67E-01	4.62E-01	8.89E-01	8.42E-01	4.62E-01	7.50E-01	5.33E-01	5.71E-01	5.38E-01	5.38E-01	34
									0.00E+00	9.38E-01	7.86E-01	8.67E-01	9.38E-01	6.43E-01	9.38E-01	7.86E-01	9.44E-01	9.38E-01	9.33E-01	35
										0.00E+00	6.15E-01	5.00E-01	8.00E-01	8.24E-01	7.14E-01	6.15E-01	4.62E-01	3.64E-01	3.00E-01	36
											0.00E+00	3.64E-01	7.14E-01	6.67E-01	6.15E-01	5.00E-01	5.71E-01	6.15E-01	5.83E-01	37
												0.00E+00	8.00E-01	7.50E-01	3.64E-01	6.15E-01	3.33E-01	5.00E-01	3.00E-01	38
													0.00E+00	8.24E-01	7.14E-01	6.15E-01	8.24E-01	8.75E-01	7.86E-01	39
														0.00E+00	8.24E-01	5.71E-01	6.25E-01	6.67E-01	8.13E-01	40
															0.00E+00	6.15E-01	5.71E-01	7.14E-01	5.83E-01	41
																0.00E+00	6.67E-01	7.14E-01	5.83E-01	42
																	0.00E+00	1.82E-01	4.17E-01	43
																		0.00E+00	4.55E-01	44
																			0.00E+00	45

Table C.7: Resemblance Matrix using the Kulczynski Coefficient - Part I

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25				
0.00E+00	8.60E-01	8.84E-01	8.57E-01	8.89E-01	8.70E-01	8.86E-01	9.13E-01	8.86E-01	9.61E-01	9.41E-01	8.91E-01	9.13E-01	8.89E-01	9.40E-01	9.35E-01	8.60E-01	8.67E-01	8.64E-01	9.15E-01	9.15E-01	8.60E-01	8.60E-01	9.15E-01	8.91E-01	8.91E-01	1		
	0.00E+00	9.13E-01	8.29E-01	9.40E-01	9.22E-01	9.39E-01	9.15E-01	9.39E-01	9.40E-01	9.20E-01	8.94E-01	9.15E-01	9.40E-01	9.18E-01	9.59E-01	9.17E-01	9.20E-01	8.05E-01	9.40E-01	9.17E-01	9.17E-01	8.91E-01	9.17E-01	9.41E-01	9.41E-01	2		
		0.00E+00	8.84E-01	9.38E-01	9.18E-01	9.11E-01	9.36E-01	9.36E-01	9.38E-01	9.17E-01	8.60E-01	9.36E-01	9.13E-01	9.15E-01	9.33E-01	9.13E-01	9.17E-01	8.89E-01	9.38E-01	9.13E-01	9.13E-01	8.86E-01	9.38E-01	9.15E-01	9.15E-01	3		
			0.00E+00	9.39E-01	9.20E-01	9.38E-01	9.38E-01	9.38E-01	9.61E-01	9.41E-01	8.64E-01	8.86E-01	9.39E-01	9.40E-01	9.58E-01	9.15E-01	9.18E-01	8.00E-01	9.15E-01	9.39E-01	9.15E-01	8.60E-01	8.89E-01	9.40E-01	9.40E-01	4		
				0.00E+00	8.44E-01	8.29E-01	8.60E-01	8.29E-01	9.17E-01	9.42E-01	9.18E-01	9.15E-01	8.33E-01	9.41E-01	8.84E-01	8.33E-01	7.75E-01	9.41E-01	9.40E-01	9.62E-01	8.00E-01	9.17E-01	9.62E-01	8.05E-01	8.05E-01	5		
					0.00E+00	8.70E-01	8.70E-01	8.41E-01	8.98E-01	9.02E-01	8.75E-01	8.96E-01	8.72E-01	9.00E-01	8.91E-01	8.14E-01	8.22E-01	9.23E-01	9.22E-01	9.64E-01	8.14E-01	8.72E-01	9.43E-01	8.75E-01	8.75E-01	6		
						0.00E+00	8.86E-01	8.57E-01	9.15E-01	9.41E-01	9.17E-01	9.13E-01	7.95E-01	9.40E-01	9.09E-01	8.60E-01	8.37E-01	9.40E-01	9.39E-01	9.39E-01	8.29E-01	9.15E-01	9.61E-01	8.00E-01	8.00E-01	7		
							0.00E+00	8.25E-01	8.89E-01	9.18E-01	9.17E-01	9.13E-01	8.89E-01	9.17E-01	8.81E-01	8.60E-01	8.67E-01	9.40E-01	9.39E-01	9.39E-01	8.60E-01	9.15E-01	9.61E-01	8.91E-01	8.91E-01	8		
								0.00E+00	9.15E-01	9.41E-01	9.17E-01	9.13E-01	8.60E-01	9.40E-01	8.81E-01	8.29E-01	8.37E-01	9.40E-01	9.39E-01	9.61E-01	8.29E-01	9.15E-01	9.61E-01	8.64E-01	8.64E-01	9		
									0.00E+00	8.10E-01	8.94E-01	8.89E-01	9.17E-01	8.05E-01	9.11E-01	9.17E-01	9.20E-01	9.62E-01	9.81E-01	9.40E-01	9.17E-01	9.40E-01	9.62E-01	9.18E-01	9.18E-01	10		
										0.00E+00	8.72E-01	9.18E-01	9.42E-01	7.44E-01	9.39E-01	9.20E-01	9.23E-01	9.43E-01	9.63E-01	9.42E-01	9.20E-01	8.96E-01	9.20E-01	9.43E-01	9.43E-01	11		
											0.00E+00	8.91E-01	9.18E-01	8.70E-01	9.38E-01	8.94E-01	8.98E-01	8.70E-01	9.18E-01	9.41E-01	8.94E-01	8.94E-01	9.20E-01	9.20E-01	9.20E-01	12		
												0.00E+00	9.15E-01	9.17E-01	9.35E-01	9.15E-01	9.18E-01	8.91E-01	9.15E-01	9.61E-01	9.15E-01	8.60E-01	8.89E-01	9.17E-01	9.17E-01	13		
													0.00E+00	9.41E-01	9.11E-01	8.64E-01	8.10E-01	9.18E-01	9.40E-01	9.40E-01	8.33E-01	9.17E-01	9.62E-01	7.69E-01	7.69E-01	14		
														0.00E+00	9.38E-01	9.18E-01	9.22E-01	9.42E-01	9.62E-01	9.41E-01	9.18E-01	8.94E-01	9.18E-01	9.42E-01	9.42E-01	15		
															0.00E+00	8.54E-01	8.89E-01	9.60E-01	9.59E-01	9.59E-01	8.84E-01	9.36E-01	9.80E-01	9.13E-01	9.13E-01	16		
																0.00E+00	8.10E-01	9.18E-01	9.40E-01	9.62E-01	8.00E-01	8.91E-01	9.40E-01	8.67E-01	8.67E-01	17		
																	0.00E+00	8.98E-01	9.42E-01	9.63E-01	7.75E-01	8.96E-01	9.42E-01	7.80E-01	7.80E-01	18		
																		0.00E+00	9.18E-01	9.41E-01	9.18E-01	8.67E-01	8.94E-01	9.20E-01	9.20E-01	19		
																			0.00E+00	9.62E-01	9.40E-01	8.64E-01	8.33E-01	9.41E-01	9.41E-01	20		
																				0.00E+00	9.62E-01	9.40E-01	9.81E-01	9.41E-01	9.41E-01	21		
																					0.00E+00	8.91E-01	9.40E-01	8.37E-01	8.37E-01	22		
																						0.00E+00	8.64E-01	9.18E-01	9.18E-01	9.18E-01	23	
																							0.00E+00	9.62E-01	9.62E-01	9.62E-01	24	
																								0.00E+00	9.62E-01	9.62E-01	9.62E-01	25

Table C.8: Resemblance Matrix using the Kulczynski Coefficient - Part II

26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	
8.64E-01	9.13E-01	8.25E-01	8.70E-01	8.84E-01	8.96E-01	9.62E-01	9.17E-01	9.62E-01	9.38E-01	9.15E-01	8.89E-01	9.15E-01	8.60E-01	8.94E-01	9.39E-01	8.29E-01	9.18E-01	9.39E-01	9.13E-01	1
9.18E-01	9.61E-01	8.60E-01	9.22E-01	8.86E-01	9.22E-01	9.41E-01	9.18E-01	9.42E-01	9.15E-01	9.40E-01	8.91E-01	9.17E-01	9.17E-01	8.41E-01	9.40E-01	8.64E-01	8.70E-01	8.91E-01	9.39E-01	2
9.15E-01	9.11E-01	8.54E-01	9.18E-01	9.09E-01	9.18E-01	9.61E-01	9.15E-01	9.40E-01	9.59E-01	9.38E-01	9.13E-01	9.38E-01	9.13E-01	9.17E-01	9.13E-01	8.25E-01	9.40E-01	9.60E-01	9.36E-01	3
9.17E-01	9.60E-01	8.25E-01	9.20E-01	8.54E-01	8.96E-01	9.62E-01	8.91E-01	9.41E-01	9.38E-01	9.15E-01	8.60E-01	8.89E-01	9.15E-01	8.37E-01	9.15E-01	8.29E-01	8.67E-01	8.89E-01	9.13E-01	4
7.69E-01	9.39E-01	9.15E-01	7.80E-01	9.38E-01	9.22E-01	9.18E-01	9.41E-01	9.63E-01	9.81E-01	9.40E-01	9.40E-01	9.62E-01	8.33E-01	9.63E-01	9.40E-01	9.17E-01	9.63E-01	9.62E-01	9.39E-01	5
8.18E-01	9.42E-01	8.96E-01	7.95E-01	9.41E-01	9.04E-01	9.00E-01	8.75E-01	9.65E-01	9.82E-01	9.22E-01	9.22E-01	9.43E-01	8.14E-01	9.45E-01	9.22E-01	8.98E-01	9.45E-01	9.64E-01	9.42E-01	6
8.33E-01	9.13E-01	9.13E-01	8.41E-01	9.36E-01	9.20E-01	9.40E-01	9.40E-01	9.41E-01	9.81E-01	9.39E-01	9.39E-01	9.61E-01	8.60E-01	9.62E-01	9.15E-01	8.89E-01	9.62E-01	9.61E-01	9.38E-01	7
8.64E-01	9.60E-01	9.13E-01	8.70E-01	9.36E-01	9.20E-01	8.91E-01	9.40E-01	9.62E-01	9.60E-01	9.39E-01	9.39E-01	9.61E-01	8.29E-01	9.62E-01	9.39E-01	9.15E-01	9.62E-01	9.61E-01	9.38E-01	8
8.33E-01	9.38E-01	9.13E-01	8.41E-01	9.36E-01	9.20E-01	9.17E-01	9.40E-01	9.62E-01	9.81E-01	9.39E-01	9.39E-01	9.61E-01	7.95E-01	9.62E-01	9.39E-01	9.15E-01	9.62E-01	9.61E-01	9.38E-01	9
9.18E-01	9.61E-01	9.61E-01	9.22E-01	9.38E-01	9.64E-01	8.37E-01	8.67E-01	9.20E-01	9.39E-01	9.81E-01	9.40E-01	9.62E-01	9.17E-01	9.82E-01	9.17E-01	9.40E-01	9.82E-01	9.81E-01	9.81E-01	10
9.22E-01	9.62E-01	9.41E-01	9.25E-01	9.40E-01	9.65E-01	8.44E-01	8.72E-01	9.00E-01	9.41E-01	9.63E-01	9.63E-01	9.20E-01	9.20E-01	9.64E-01	9.20E-01	9.44E-01	9.63E-01	9.62E-01	9.62E-01	11
8.96E-01	9.40E-01	8.64E-01	8.75E-01	8.60E-01	9.23E-01	9.20E-01	8.96E-01	8.98E-01	9.40E-01	9.18E-01	8.67E-01	8.94E-01	8.94E-01	8.98E-01	8.67E-01	8.37E-01	9.22E-01	9.41E-01	9.40E-01	12
9.17E-01	9.81E-01	8.86E-01	9.20E-01	8.54E-01	8.96E-01	9.17E-01	8.33E-01	9.18E-01	9.38E-01	9.15E-01	8.60E-01	8.89E-01	9.15E-01	9.18E-01	8.89E-01	8.89E-01	9.18E-01	9.15E-01	9.13E-01	13
8.37E-01	9.15E-01	9.15E-01	8.14E-01	9.38E-01	9.22E-01	9.41E-01	9.41E-01	9.42E-01	9.81E-01	9.40E-01	9.40E-01	9.62E-01	8.64E-01	9.63E-01	9.17E-01	8.91E-01	9.63E-01	9.62E-01	9.39E-01	14
9.20E-01	9.62E-01	9.40E-01	9.23E-01	9.39E-01	9.64E-01	8.70E-01	8.70E-01	8.98E-01	9.40E-01	9.62E-01	8.94E-01	9.18E-01	9.18E-01	9.64E-01	9.18E-01	9.18E-01	9.43E-01	9.62E-01	9.62E-01	15
8.86E-01	9.58E-01	9.35E-01	8.91E-01	9.57E-01	9.40E-01	9.13E-01	9.38E-01	9.81E-01	9.80E-01	9.59E-01	9.36E-01	9.80E-01	8.84E-01	9.81E-01	9.59E-01	9.36E-01	9.81E-01	9.80E-01	9.58E-01	16
8.05E-01	9.39E-01	8.89E-01	8.14E-01	9.38E-01	8.98E-01	9.18E-01	9.18E-01	9.63E-01	9.81E-01	9.40E-01	8.91E-01	9.40E-01	8.00E-01	9.42E-01	9.17E-01	8.91E-01	9.42E-01	9.62E-01	9.39E-01	17
7.44E-01	9.41E-01	8.94E-01	7.18E-01	9.40E-01	9.02E-01	9.22E-01	9.22E-01	9.64E-01	9.82E-01	9.42E-01	9.20E-01	9.42E-01	8.10E-01	9.44E-01	9.20E-01	8.96E-01	9.44E-01	9.63E-01	9.41E-01	18
9.20E-01	9.62E-01	8.33E-01	9.00E-01	8.60E-01	9.00E-01	9.63E-01	8.96E-01	9.22E-01	9.40E-01	9.18E-01	8.67E-01	8.94E-01	9.18E-01	8.14E-01	9.18E-01	8.37E-01	8.44E-01	8.67E-01	9.17E-01	19
9.41E-01	9.81E-01	8.89E-01	9.22E-01	9.13E-01	8.72E-01	9.82E-01	9.41E-01	8.96E-01	9.81E-01	7.63E-01	8.91E-01	8.64E-01	9.40E-01	9.42E-01	9.17E-01	8.91E-01	8.41E-01	8.33E-01	8.29E-01	20
9.62E-01	8.60E-01	9.15E-01	9.64E-01	9.38E-01	9.22E-01	9.62E-01	9.62E-01	9.82E-01	8.29E-01	9.62E-01	9.62E-01	9.81E-01	9.62E-01	8.96E-01	9.81E-01	9.81E-01	9.82E-01	9.81E-01	9.61E-01	21
7.69E-01	9.39E-01	8.89E-01	7.80E-01	9.38E-01	8.98E-01	9.18E-01	9.18E-01	9.63E-01	9.81E-01	9.40E-01	9.17E-01	9.40E-01	8.00E-01	9.42E-01	9.17E-01	8.91E-01	9.42E-01	9.62E-01	9.39E-01	22
8.94E-01	9.61E-01	8.29E-01	8.98E-01	8.86E-01	8.72E-01	9.62E-01	8.67E-01	9.20E-01	9.61E-01	8.64E-01	8.33E-01	8.64E-01	8.91E-01	8.96E-01	9.17E-01	8.33E-01	8.70E-01	8.91E-01	8.60E-01	23
9.41E-01	1.00E+00	8.89E-01	9.43E-01	8.86E-01	8.72E-01	9.62E-01	8.94E-01	8.70E-01	9.61E-01	8.33E-01	8.33E-01	8.00E-01	9.40E-01	9.20E-01	8.64E-01	8.91E-01	8.10E-01	8.33E-01	8.60E-01	24
8.10E-01	9.17E-01	9.17E-01	7.86E-01	9.39E-01	9.23E-01	9.42E-01	9.42E-01	9.43E-01	9.81E-01	9.41E-01	9.41E-01	9.62E-01	8.67E-01	9.64E-01	9.18E-01	8.94E-01	9.64E-01	9.62E-01	9.40E-01	25
0.00E+00	9.40E-01	8.91E-01	7.50E-01	9.39E-01	9.00E-01	9.20E-01	9.20E-01	9.64E-01	9.81E-01	9.41E-01	9.18E-01	9.41E-01	8.05E-01	9.43E-01	9.18E-01	8.94E-01	9.43E-01	9.62E-01	9.40E-01	26
	0.00E+00	9.38E-01	9.42E-01	9.59E-01	9.20E-01	9.81E-01	9.81E-01	9.82E-01	9.13E-01	9.81E-01	9.81E-01	1.00E+00	9.39E-01	8.94E-01	9.81E-01	9.15E-01	1.00E+00	1.00E+00	9.81E-01	27
		0.00E+00	8.96E-01	8.54E-01	8.70E-01	9.62E-01	8.91E-01	9.41E-01	9.38E-01	8.89E-01	8.60E-01	8.89E-01	8.89E-01	8.67E-01	9.15E-01	7.95E-01	8.94E-01	9.15E-01	8.86E-01	28
			0.00E+00	9.41E-01	9.04E-01	9.23E-01	9.23E-01	9.65E-01	9.82E-01	9.22E-01	9.22E-01	9.43E-01	8.14E-01	9.45E-01	9.22E-01	8.98E-01	9.45E-01	9.64E-01	9.42E-01	29
				0.00E+00	9.18E-01	9.39E-01	8.89E-01	9.17E-01	9.11E-01	9.13E-01	8.57E-01	8.86E-01	9.38E-01	8.91E-01	9.13E-01	8.57E-01	9.17E-01	9.13E-01	9.11E-01	30
					0.00E+00	9.64E-01	9.00E-01	9.45E-01	9.42E-01	8.72E-01	8.98E-01	8.98E-01	8.98E-01	8.51E-01	8.98E-01	8.72E-01	8.78E-01	8.98E-01	8.96E-01	31
						0.00E+00	8.96E-01	9.43E-01	9.40E-01	9.82E-01	9.41E-01	9.62E-01	9.18E-01	9.82E-01	9.41E-01	9.62E-01	9.82E-01	9.82E-01	9.81E-01	32
							0.00E+00	9.22E-01	9.40E-01	9.41E-01	8.67E-01	8.94E-01	9.18E-01	9.22E-01	8.94E-01	8.94E-01	9.22E-01	9.41E-01	9.40E-01	33
								0.00E+00	9.82E-01	8.96E-01	8.96E-01	8.41E-01	9.63E-01	9.44E-01	8.41E-01	9.20E-01	8.48E-01	8.70E-01	8.67E-01	34
									0.00E+00	9.81E-01	9.39E-01	9.61E-01	9.81E-01	8.94E-01	9.81E-01	9.39E-01	9.82E-01	9.81E-01	9.81E-01	35
										0.00E+00	8.91E-01	8.64E-01	9.40E-01	9.42E-01	9.17E-01	8.91E-01	8.41E-01	8.33E-01	8.29E-01	36
											0.00E+00	8.33E-01	9.17E-01	8.96E-01	8.91E-01	8.64E-01	8.70E-01	8.91E-01	8.89E-01	37
												0.00E+00	9.40E-01	9.20E-01	8.33E-01	8.91E-01	8.10E-01	8.64E-01	8.29E-01	38
													0.00E+00	9.42E-01	9.17E-01	8.91E-01	9.42E-01	9.62E-01	9.39E-01	39
														0.00E+00	9.42E-01	8.70E-01	8.75E-01	8.96E-01	9.41E-01	40
															0.00E+00	8.91E-01	8.70E-01	9.17E-01	8.89E-01	41
																0.00E+00	8.96E-01	9.17E-01	8.89E-01	42
																	0.00E+00	7.75E-01	8.37E-01	43
																		0.00E+00	8.60E-01	44
																			0.00E+00	45

Table C.9: Resemblance Matrix using the Sokal-Sneath Coefficient - Part I

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	
0.00E+00	6.25E-01	6.67E-01	5.71E-01	7.37E-01	7.27E-01	7.06E-01	8.00E-01	7.06E-01	9.29E-01	8.97E-01	7.62E-01	8.00E-01	7.37E-01	8.89E-01	8.42E-01	6.25E-01	7.00E-01	6.67E-01	8.18E-01	8.18E-01	6.25E-01	6.25E-01	8.18E-01	7.62E-01	1
	0.00E+00	8.00E-01	4.62E-01	8.89E-01	8.67E-01	8.80E-01	8.18E-01	8.80E-01	8.89E-01	8.57E-01	7.83E-01	8.18E-01	8.89E-01	8.46E-01	9.17E-01	8.33E-01	8.57E-01	4.29E-01	8.89E-01	8.33E-01	8.33E-01	7.62E-01	8.33E-01	8.97E-01	2
		0.00E+00	6.67E-01	8.70E-01	8.46E-01	7.78E-01	8.57E-01	8.57E-01	8.70E-01	8.33E-01	6.25E-01	8.57E-01	8.00E-01	8.18E-01	8.24E-01	8.00E-01	8.33E-01	7.37E-01	8.70E-01	8.00E-01	8.00E-01	7.06E-01	8.70E-01	8.18E-01	3
			0.00E+00	8.80E-01	8.57E-01	8.70E-01	8.70E-01	8.70E-01	9.29E-01	8.97E-01	6.67E-01	7.06E-01	8.80E-01	8.89E-01	9.09E-01	8.18E-01	8.46E-01	3.33E-01	8.18E-01	8.80E-01	8.18E-01	6.25E-01	7.37E-01	8.89E-01	4
				0.00E+00	6.67E-01	4.62E-01	6.25E-01	4.62E-01	8.33E-01	9.03E-01	8.46E-01	8.18E-01	5.33E-01	8.97E-01	6.67E-01	5.33E-01	3.08E-01	8.97E-01	8.89E-01	9.33E-01	3.33E-01	8.33E-01	9.33E-01	4.29E-01	5
					0.00E+00	7.27E-01	7.27E-01	6.32E-01	8.15E-01	8.39E-01	7.69E-01	8.00E-01	7.50E-01	8.28E-01	7.62E-01	5.56E-01	6.36E-01	8.75E-01	8.67E-01	9.44E-01	5.56E-01	7.50E-01	9.09E-01	7.69E-01	6
						0.00E+00	7.06E-01	5.71E-01	8.18E-01	8.97E-01	8.33E-01	8.00E-01	2.00E-01	8.89E-01	7.50E-01	6.25E-01	5.88E-01	8.89E-01	8.80E-01	8.80E-01	4.62E-01	8.18E-01	9.29E-01	3.33E-01	7
							0.00E+00	3.64E-01	7.37E-01	8.46E-01	8.33E-01	8.00E-01	7.37E-01	8.33E-01	6.15E-01	6.25E-01	7.00E-01	8.89E-01	8.80E-01	8.80E-01	6.25E-01	8.18E-01	9.29E-01	7.62E-01	8
								0.00E+00	8.18E-01	8.97E-01	8.33E-01	8.00E-01	6.25E-01	8.89E-01	6.15E-01	4.62E-01	5.88E-01	8.89E-01	8.80E-01	9.29E-01	4.62E-01	8.18E-01	9.29E-01	6.67E-01	9
									0.00E+00	5.00E-01	7.83E-01	7.37E-01	8.33E-01	4.29E-01	7.78E-01	8.33E-01	8.57E-01	9.38E-01	9.70E-01	8.89E-01	8.33E-01	8.89E-01	9.33E-01	8.46E-01	10
										0.00E+00	7.50E-01	8.46E-01	9.03E-01	1.67E-01	8.80E-01	8.57E-01	8.75E-01	9.09E-01	9.41E-01	9.03E-01	8.57E-01	8.00E-01	8.57E-01	9.09E-01	11
											0.00E+00	7.62E-01	8.46E-01	7.27E-01	8.70E-01	7.83E-01	8.15E-01	7.27E-01	8.46E-01	8.97E-01	7.83E-01	7.83E-01	7.83E-01	8.57E-01	12
												0.00E+00	8.18E-01	8.33E-01	8.42E-01	8.18E-01	8.46E-01	7.62E-01	8.18E-01	9.29E-01	8.18E-01	6.25E-01	7.37E-01	8.33E-01	13
													0.00E+00	8.97E-01	7.78E-01	6.67E-01	5.00E-01	8.46E-01	8.89E-01	8.89E-01	5.33E-01	8.33E-01	9.33E-01	1.82E-01	14
														0.00E+00	8.70E-01	8.46E-01	8.67E-01	9.03E-01	9.38E-01	8.97E-01	8.46E-01	7.83E-01	8.46E-01	9.03E-01	15
															0.00E+00	5.00E-01	7.37E-01	9.23E-01	9.17E-01	9.17E-01	6.67E-01	8.57E-01	9.63E-01	8.00E-01	16
																0.00E+00	5.00E-01	8.46E-01	8.89E-01	9.33E-01	3.33E-01	7.62E-01	8.89E-01	7.00E-01	17
																	0.00E+00	8.15E-01	9.03E-01	9.41E-01	3.08E-01	8.00E-01	9.03E-01	4.00E-01	18
																		0.00E+00	8.46E-01	8.97E-01	8.46E-01	7.00E-01	7.83E-01	8.57E-01	19
																			0.00E+00	9.33E-01	8.89E-01	6.67E-01	5.33E-01	8.97E-01	20
																				0.00E+00	9.33E-01	8.89E-01	9.70E-01	8.97E-01	21
																					0.00E+00	7.62E-01	8.89E-01	5.88E-01	22
																						0.00E+00	6.67E-01	8.46E-01	23
																							0.00E+00	9.38E-01	24
																								0.00E+00	25

Table C.10: Resemblance Matrix using the SokalSneath Coefficient - Part II

26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	
6.67E-01	8.00E-01	3.64E-01	7.27E-01	6.67E-01	8.00E-01	9.33E-01	8.33E-01	9.38E-01	8.70E-01	8.18E-01	7.37E-01	8.18E-01	6.25E-01	7.83E-01	8.80E-01	4.62E-01	8.46E-01	8.80E-01	8.00E-01	1
8.46E-01	9.29E-01	6.25E-01	8.67E-01	7.06E-01	8.67E-01	8.97E-01	8.46E-01	9.03E-01	8.18E-01	8.89E-01	7.62E-01	8.33E-01	8.33E-01	6.32E-01	8.89E-01	6.67E-01	7.27E-01	7.62E-01	8.80E-01	2
8.18E-01	7.78E-01	5.00E-01	8.46E-01	7.50E-01	8.46E-01	9.29E-01	8.18E-01	8.89E-01	9.17E-01	8.70E-01	8.00E-01	8.70E-01	8.00E-01	8.33E-01	8.00E-01	3.64E-01	8.89E-01	9.23E-01	8.57E-01	3
8.33E-01	9.23E-01	3.64E-01	8.57E-01	5.00E-01	8.00E-01	9.33E-01	7.62E-01	8.97E-01	8.70E-01	8.18E-01	6.25E-01	7.37E-01	8.18E-01	5.88E-01	8.18E-01	4.62E-01	7.00E-01	7.37E-01	8.00E-01	4
1.82E-01	8.80E-01	8.18E-01	4.00E-01	8.70E-01	8.67E-01	8.46E-01	8.97E-01	9.41E-01	9.68E-01	8.89E-01	8.89E-01	9.33E-01	5.33E-01	9.41E-01	8.89E-01	8.33E-01	9.41E-01	9.33E-01	8.80E-01	5
6.00E-01	9.03E-01	8.00E-01	5.71E-01	8.97E-01	8.48E-01	8.28E-01	7.69E-01	9.50E-01	9.73E-01	8.67E-01	8.67E-01	9.09E-01	5.56E-01	9.19E-01	8.67E-01	8.15E-01	9.19E-01	9.44E-01	9.03E-01	6
5.33E-01	8.00E-01	8.00E-01	6.32E-01	8.57E-01	8.57E-01	8.89E-01	8.89E-01	8.97E-01	9.66E-01	8.80E-01	8.80E-01	9.29E-01	6.25E-01	9.38E-01	8.18E-01	7.37E-01	9.38E-01	9.29E-01	8.70E-01	7
6.67E-01	9.23E-01	8.00E-01	7.27E-01	8.57E-01	8.57E-01	7.62E-01	8.89E-01	9.38E-01	9.23E-01	8.80E-01	8.80E-01	9.29E-01	4.62E-01	9.38E-01	8.80E-01	8.18E-01	9.38E-01	9.29E-01	8.70E-01	8
5.33E-01	8.70E-01	8.00E-01	6.32E-01	8.57E-01	8.57E-01	8.33E-01	8.89E-01	9.38E-01	9.66E-01	8.80E-01	8.80E-01	9.29E-01	2.00E-01	9.38E-01	8.80E-01	8.18E-01	9.38E-01	9.29E-01	8.70E-01	9
8.46E-01	9.29E-01	9.29E-01	8.67E-01	8.70E-01	9.44E-01	5.88E-01	7.00E-01	8.57E-01	8.80E-01	9.70E-01	8.89E-01	9.33E-01	8.33E-01	9.73E-01	8.33E-01	8.89E-01	9.73E-01	9.70E-01	9.68E-01	10
8.67E-01	9.38E-01	8.97E-01	8.82E-01	8.89E-01	9.50E-01	6.67E-01	7.50E-01	8.28E-01	8.97E-01	9.41E-01	8.00E-01	8.57E-01	8.57E-01	9.47E-01	8.57E-01	9.14E-01	9.14E-01	9.41E-01	9.38E-01	11
8.00E-01	8.89E-01	6.67E-01	7.69E-01	6.25E-01	8.75E-01	8.57E-01	8.00E-01	8.15E-01	8.89E-01	8.46E-01	7.00E-01	7.83E-01	7.83E-01	8.15E-01	7.00E-01	5.88E-01	8.67E-01	8.97E-01	8.89E-01	12
8.33E-01	9.66E-01	7.06E-01	8.57E-01	5.00E-01	8.00E-01	8.33E-01	5.33E-01	8.46E-01	8.70E-01	8.18E-01	6.25E-01	7.37E-01	8.18E-01	8.46E-01	7.37E-01	7.37E-01	8.46E-01	8.18E-01	8.00E-01	13
5.88E-01	8.18E-01	8.18E-01	5.56E-01	8.70E-01	8.67E-01	8.97E-01	8.97E-01	9.03E-01	9.68E-01	8.89E-01	8.89E-01	9.33E-01	6.67E-01	9.41E-01	8.33E-01	7.62E-01	9.41E-01	9.33E-01	8.80E-01	14
8.57E-01	9.33E-01	8.89E-01	8.75E-01	8.80E-01	9.47E-01	7.27E-01	7.27E-01	8.15E-01	8.89E-01	9.38E-01	7.83E-01	8.46E-01	8.46E-01	9.44E-01	8.46E-01	8.46E-01	9.09E-01	9.38E-01	9.33E-01	15
7.06E-01	9.09E-01	8.42E-01	7.62E-01	9.00E-01	8.89E-01	8.00E-01	8.70E-01	9.68E-01	9.60E-01	9.17E-01	8.57E-01	9.63E-01	6.67E-01	9.68E-01	9.17E-01	8.57E-01	9.68E-01	9.63E-01	9.09E-01	16
4.29E-01	8.80E-01	7.37E-01	5.56E-01	8.70E-01	8.15E-01	8.46E-01	8.46E-01	9.41E-01	9.68E-01	8.89E-01	7.62E-01	8.89E-01	3.33E-01	9.03E-01	8.33E-01	7.62E-01	9.03E-01	9.33E-01	8.80E-01	17
1.67E-01	8.97E-01	7.83E-01	1.54E-01	8.89E-01	8.39E-01	8.67E-01	8.67E-01	9.47E-01	9.71E-01	9.03E-01	8.57E-01	9.03E-01	5.00E-01	9.14E-01	8.57E-01	8.00E-01	9.14E-01	9.41E-01	8.97E-01	18
8.57E-01	9.33E-01	5.33E-01	8.28E-01	6.25E-01	8.28E-01	9.41E-01	8.00E-01	8.67E-01	8.89E-01	8.46E-01	7.00E-01	7.83E-01	8.46E-01	5.56E-01	8.46E-01	5.88E-01	6.67E-01	7.00E-01	8.33E-01	19
8.97E-01	9.68E-01	7.37E-01	8.67E-01	8.00E-01	7.50E-01	9.71E-01	8.97E-01	8.00E-01	9.68E-01	0.00E+00	7.62E-01	6.67E-01	8.89E-01	9.03E-01	8.33E-01	7.62E-01	6.32E-01	5.33E-01	4.62E-01	20
9.38E-01	6.25E-01	8.18E-01	9.44E-01	8.70E-01	8.67E-01	9.38E-01	9.38E-01	9.73E-01	4.62E-01	9.33E-01	9.33E-01	9.70E-01	9.33E-01	8.00E-01	9.70E-01	7.62E-01	9.73E-01	9.70E-01	9.29E-01	21
1.82E-01	8.80E-01	7.37E-01	4.00E-01	8.70E-01	8.15E-01	8.46E-01	8.46E-01	9.41E-01	9.68E-01	8.89E-01	8.33E-01	8.89E-01	3.33E-01	9.03E-01	8.33E-01	7.62E-01	9.03E-01	9.33E-01	8.80E-01	22
7.83E-01	9.29E-01	4.62E-01	8.15E-01	7.06E-01	7.50E-01	9.38E-01	7.00E-01	8.57E-01	9.29E-01	6.67E-01	5.33E-01	6.67E-01	7.62E-01	8.00E-01	8.33E-01	5.33E-01	7.27E-01	7.62E-01	6.25E-01	23
8.97E-01	1.00E+00	7.37E-01	9.09E-01	7.06E-01	7.50E-01	9.38E-01	7.83E-01	7.27E-01	9.29E-01	5.33E-01	5.33E-01	3.33E-01	8.89E-01	8.57E-01	6.67E-01	7.62E-01	5.00E-01	5.33E-01	6.25E-01	24
5.00E-01	8.33E-01	8.33E-01	4.71E-01	8.80E-01	8.75E-01	9.03E-01	9.03E-01	9.09E-01	9.70E-01	8.97E-01	8.97E-01	9.38E-01	7.00E-01	9.44E-01	8.46E-01	7.83E-01	9.44E-01	9.38E-01	8.89E-01	25
0.00E+00	8.89E-01	7.62E-01	2.86E-01	8.80E-01	8.28E-01	8.57E-01	8.57E-01	9.44E-01	9.70E-01	8.97E-01	8.46E-01	8.97E-01	4.29E-01	9.09E-01	8.46E-01	7.83E-01	9.09E-01	9.38E-01	8.89E-01	26
	0.00E+00	8.70E-01	9.03E-01	9.17E-01	8.57E-01	9.70E-01	9.70E-01	9.71E-01	8.00E-01	9.68E-01	9.68E-01	1.00E+00	8.80E-01	7.83E-01	9.68E-01	8.18E-01	1.00E+00	1.00E+00	9.66E-01	27
		0.00E+00	8.00E-01	5.00E-01	7.27E-01	9.33E-01	7.62E-01	8.97E-01	8.70E-01	7.37E-01	6.25E-01	7.37E-01	7.37E-01	7.00E-01	8.18E-01	2.00E-01	7.83E-01	8.18E-01	7.06E-01	28
			0.00E+00	8.97E-01	8.48E-01	8.75E-01	8.75E-01	9.50E-01	9.73E-01	8.67E-01	8.67E-01	9.09E-01	5.56E-01	9.19E-01	8.67E-01	8.15E-01	9.19E-01	9.44E-01	9.03E-01	29
				0.00E+00	8.46E-01	8.80E-01	7.37E-01	8.33E-01	7.78E-01	8.00E-01	5.71E-01	7.06E-01	8.70E-01	7.62E-01	8.00E-01	5.71E-01	8.33E-01	8.00E-01	7.78E-01	30
					0.00E+00	9.47E-01	8.28E-01	9.19E-01	9.03E-01	7.50E-01	8.15E-01	8.15E-01	8.15E-01	7.20E-01	8.15E-01	7.50E-01	7.86E-01	8.15E-01	8.00E-01	31
						0.00E+00	8.00E-01	9.09E-01	8.89E-01	9.71E-01	8.97E-01	9.38E-01	8.46E-01	9.74E-01	8.97E-01	9.38E-01	9.74E-01	9.71E-01	9.70E-01	32
							0.00E+00	8.67E-01	8.89E-01	8.97E-01	7.00E-01	7.83E-01	8.46E-01	8.67E-01	7.83E-01	7.83E-01	8.67E-01	8.97E-01	8.89E-01	33
								0.00E+00	9.71E-01	8.00E-01	6.32E-01	9.41E-01	9.14E-01	6.32E-01	8.57E-01	6.96E-01	7.27E-01	7.27E-01	7.00E-01	34
									0.00E+00	9.68E-01	8.80E-01	9.29E-01	9.68E-01	7.83E-01	9.68E-01	8.80E-01	9.71E-01	9.68E-01	9.66E-01	35
										0.00E+00	7.62E-01	6.67E-01	8.89E-01	9.03E-01	8.33E-01	7.62E-01	6.32E-01	5.33E-01	4.62E-01	36
											0.00E+00	5.33E-01	8.33E-01	8.00E-01	7.62E-01	6.67E-01	7.27E-01	7.62E-01	7.37E-01	37
												0.00E+00	8.89E-01	8.57E-01	5.33E-01	7.62E-01	5.00E-01	6.67E-01	4.62E-01	38
													0.00E+00	9.03E-01	8.33E-01	7.62E-01	9.03E-01	9.33E-01	8.80E-01	39
														0.00E+00	7.27E-01	7.69E-01	8.00E-01	8.97E-01	8.00E-01	40
															0.00E+00	7.62E-01	7.27E-01	8.33E-01	7.37E-01	41
																0.00E+00	8.00E-01	8.33E-01	7.37E-01	42
																	0.00E+00	3.08E-01	5.88E-01	43
																		0.00E+00	6.25E-01	44
																			0.00E+00	45

Appendix D

TML Transformations

In this appendix, the complete TML transformations are provided.

Listing D.1: TML2ECore Transformation

```
1 pre {
2   var traceModel := new ECore!EClass;
3   traceModel.name := 'TraceModel';
4   var traceContext := new ECore!EClass;
5   traceContext.name := 'TraceContext';
6   traceContext.`abstract` := true;
7   var contextsReference :=new ECore!EReference;
8   contextsReference.name := 'contexts';
9   contextsReference.eType := traceContext;
10  contextsReference.containment := true;
11  contextsReference.upperBound := -1;
12  traceModel.eStructuralFeatures.add(contextsReference);
13  var traceLink := new ECore!EClass;
14  traceLink.name := 'TraceLink';
15  traceLink.`abstract` := true;
16  var traceLinkEnd := new ECore!EClass;
17  traceLinkEnd.name := 'TraceLinkEnd';
18  traceLinkEnd.`abstract` := true;
19  var modelLinksReference := new ECore!EReference;
20  modelLinksReference.name := 'links';
21  modelLinksReference.eType := traceLink;
22  modelLinksReference.containment := true;
23  modelLinksReference.upperBound := -1;
24  traceModel.eStructuralFeatures.add(modelLinksReference);
```

```

25  var traceLinkEndStatus = new ECore!EClass;
26  traceLinkEndStatus.name = "TraceLinkEndStatus";
27  traceLinkEndStatus.`abstract` = true;
28  var traceLinkEndOKStatus = new ECore!EClass;
29  traceLinkEndOKStatus.name = "TraceLinkEndOKStatus";
30  traceLinkEndOKStatus.eSuperTypes.add(traceLinkEndStatus);
31  var traceLinkEndInvalidStatus = new ECore!EClass;
32  traceLinkEndInvalidStatus.name = "TraceLinkEndInvalidStatus";
33  traceLinkEndInvalidStatus.eSuperTypes.add(traceLinkEndStatus);
34  var traceLinkEndAmbiguousStatus = new ECore!EClass;
35  traceLinkEndAmbiguousStatus.name = "TraceLinkEndAmbiguousStatus";
36  traceLinkEndAmbiguousStatus.eSuperTypes.add(traceLinkEndStatus);
37  var traceLinkEndAmbiguousStatusCandidates = new ECore!EReference;
38  traceLinkEndAmbiguousStatusCandidates.name = "candidates";
39  traceLinkEndAmbiguousStatusCandidates.upperBound = -1;
40  traceLinkEndAmbiguousStatusCandidates.lowerBound = 2;
41  traceLinkEndAmbiguousStatus.eStructuralFeatures.add(
    traceLinkEndAmbiguousStatusCandidates);
42  traceLinkEndAmbiguousStatusCandidates.eType = EcoreM2!EClass.all.
    selectOne(c|c.name = "EObject");
43  var traceLinkEndStatusReference = new ECore!EReference;
44  traceLinkEndStatusReference.name = "status";
45  traceLinkEndStatusReference.eType = traceLinkEndStatus;
46  traceLinkEndStatusReference.lowerBound = 1;
47  traceLinkEndStatusReference.lowerBound = 1;
48  traceLinkEnd.eStructuralFeatures.add(traceLinkEndStatusReference);
49  traceLinkEndStatusReference.containment = true;
50 }
51
52 rule Trace2Package
53   transform s : Trace!Trace
54   to t : ECore!EPackage {
55     t.name := s.name;
56     t.nsURI := s.name;
57     t.nsPrefix := s.name;
58     t.eClassifiers.add(traceModel);
59     t.eClassifiers.add(traceLink);
60     t.eClassifiers.add(traceContext);
61     t.eClassifiers.add(traceLinkEnd);
62     t.eClassifiers.add(traceLinkEndStatus);
63     t.eClassifiers.add(traceLinkEndOKStatus);

```

```

64     t.eClassifiers.add(traceLinkEndInvalidStatus);
65     t.eClassifiers.add(traceLinkEndAmbiguousStatus);
66     t.eClassifiers.addAll(s.links.equivalent());
67     t.eClassifiers.addAll(s.contexts.equivalent());
68     for (c in s.links) {
69         t.eClassifiers.addAll(c.ends.equivalent());
70     }
71 }
72
73 rule TraceLink2EClass
74 transform s : Trace!TraceLink
75 to t : ECore!EClass {
76     t.name := s.name + 'TraceLink';
77     t.eSuperTypes.add(traceLink);
78     for (c in s.contexts) {
79         var ref := new ECore!EReference;
80         ref.eType := c.equivalent();
81         ref.name := c.name.firstToLowerCase();
82         ref.upperBound := -1;
83         t.eStructuralFeatures.add(ref);
84     }
85     for (c in s.ends) {
86         var ref := new ECore!EReference;
87         ref.eType := c.equivalent();
88         ref.name := c.name;
89         ref.lowerBound := c.lowerBound;
90         ref.upperBound := c.upperBound;
91         ref.containment := true;
92         t.eStructuralFeatures.add(ref);
93     }
94 }
95
96 rule Context2EClass
97 transform s : Trace!Context
98 to t : ECore!EClass {
99     t.eSuperTypes.add(traceContext);
100    t.name := s.name + 'Context';
101    t.eStructuralFeatures.addAll(s.data.equivalent());
102 }
103
104 rule ContextData2EAttribute

```

```

105  transform s : Trace!ContextData
106  to t : ECore!EAttribute {
107      t.name := s.name;
108      t.eType := s.type.literal.createEType();
109  }
110
111  rule MaintenanceData2EAttribute
112  transform s : Trace!MaintenanceData
113  to t : ECore!EAttribute {
114      guard : s.type.literal <> 'Set'
115      t.name := s.name;
116      t.eType := s.type.literal.createEType();
117  }
118
119  rule TraceLinkEnd2EClass
120  transform s : Trace!TraceLinkEnd
121  to t : ECore!EClass {
122      t.name := s.eContainer().name + s.name.firstToUpperCase() + '
          LinkEnd';
123      t.eSuperTypes.add(traceLinkEnd);
124      t.eStructuralFeatures.addAll(s.maintenanceData.equivalent());
125
126      var ref := new ECore!EReference;
127      ref.eType := s.type;
128      ref.name := "target";
129      ref.lowerBound := 1;
130      ref.upperBound := 1;
131      t.eStructuralFeatures.add(ref);
132  }
133
134  @cached
135  operation String createEType() {
136      var type := new ECore!EDataType;
137      type.name := self;
138      type.instanceClassName := 'java.lang.' + self;
139      Trace!Trace.all.first().equivalent().eClassifiers.add(type);
140      return type;
141  }
142
143  post {
144      traceLink.eSuperTypes.addAll(Trace!Context.all.select(c|c.`default`))

```

```
    .equivalent());
145 }
```

Listing D.2: TML2EVL Transformation

```
1 [%for (link in Trace!TraceLink.allInstances) { %]
2   [%for (end in link.ends.select(e|e.forAll)) { %]
3     context [%=end.type.name%] {
4       constraint OneForEach[%=end.name.firstToUpperCase()%]{
5         [%=end.computeGuard()%]
6         [%if (end.isMany()) {%]
7           check : [%=link.name%]TraceLink.all.exists(e|e.[%=end.name%].
8             target.includes(self))
9           [%} else {%]
10          check : [%=link.name%]TraceLink.all.exists(e|e.[%=end.name%].
11            target = self)
12          [%}%]
13          message : 'No links of type [%=link.name%] found for [%=end.name%]
14            ' + self
15        }
16      }
17    }
18  [%}%]
19 [%for(end in link.ends.select(e|e.unique)) { %]
20   context [%=end.type.name%] {
21     constraint Unique[%=end.name.firstToUpperCase()%]{
22       [%=end.computeGuard()%]
23       [%if (end.isMany()) {%]
24         check : [%=link.name%]TraceLink.all.select(e|e.[%=end.name%].
25           target.includes(self)).size() < 2
26         [%} else {%]
27         check : [%=link.name%]TraceLink.all.select(e|e.[%=end.name%].
28           target = self).size() < 2
29         [%}%]
30         message : 'Multiple links of type [%=link.name%] found for [%=end.
31           name%] ' + self
32       }
33     }
34   }
35 [%}%]
```

```

31 [%for (context in Trace!Context.allInstances) { %]
32   context [%=context.name%]Context {
33     [%for (data in context.data) { %]
34       [%=data.getInvariantType()%] NotOptional[%=data.name%] {
35         check : self.[%=data.name%].isDefined()
36         [%if (data.type.literal = 'String'){%]
37           and self.[%=data.name%].trim() <> ''
38         [%}%]
39         message : 'No value specified for attribute [%=data.name%]'
40       }
41     [%}%]
42   }
43 [%}%]
44
45 [%
46 operation Trace!TraceLinkEnd computeGuard() : String {
47   if (self.ofTypeOnly) {
48     return 'guard : self.isTypeOf(' + self.type.name + ')';
49   }
50   return '';
51 }
52 operation Trace!TraceLinkEnd checkDefined() : String {
53   if (self.upperBound <= 1 and self.upperBound <> -1) { return '.
54     isDefined()'; }
55   else { return '.isEmpty()'; }
56 }
57 operation Trace!ContextData getInvariantType() : String {
58   if (self.optional = true) { return 'critique'; }
59   else { return 'constraint'; }
60 }
61 operation Trace!TraceLinkEnd isMany() : Boolean {
62   return self.upperBound > 1 or self.upperBound = -1;
63 }
64 %]

```

Appendix E

EVL Constraints for the Filesystem Case-Study

In this appendix, the EVL constraints, which are used in the *Filesystem* case-study, are provided.

Listing E.1: EVL Constraints for the Filesystem Case-Study

```
1 context EPackage {
2   constraint OneForEachEPackage{
3
4     guard : self.isTypeOf(EPackage)
5
6     check : EPackage2CanvasTraceLink.all.exists(e|e.EPackage.target =
          self)
7
8     message : 'No links of type EPackage2Canvas found for EPackage ' +
          self
9   }
10 }
11 context Canvas {
12   constraint OneForEachCanvas{
13
14     guard : self.isTypeOf(Canvas)
15
16     check : EPackage2CanvasTraceLink.all.exists(e|e.Canvas.target =
          self)
17
18     message : 'No links of type EPackage2Canvas found for Canvas ' +
```

```

        self
19  }
20 }
21 context EPackage {
22   constraint UniqueEPackage{
23
24     guard : self.isTypeOf(EPackage)
25
26     check : EPackage2CanvasTraceLink.all.select(e|e.EPackage.target =
        self).size() < 2
27
28     message : 'Multiple links of type EPackage2Canvas found for
        EPackage ' + self
29   }
30 }
31 context Canvas {
32   constraint UniqueCanvas{
33
34     guard : self.isTypeOf(Canvas)
35
36     check : EPackage2CanvasTraceLink.all.select(e|e.Canvas.target =
        self).size() < 2
37
38     message : 'Multiple links of type EPackage2Canvas found for Canvas
        ' + self
39   }
40 }
41 context EClass {
42   constraint OneForEachEClass{
43
44     guard : self.isTypeOf(EClass)
45
46     check : EClass2NodeTraceLink.all.exists(e|e.EClass.target = self)
47
48     message : 'No links of type EClass2Node found for EClass ' + self
49   }
50 }
51 context Node {
52   constraint OneForEachNode{
53
54     guard : self.isTypeOf(Node)

```

```

55
56     check : EClass2NodeTraceLink.all.exists(e|e.Node.target = self)
57
58     message : 'No links of type EClass2Node found for Node ' + self
59 }
60 }
61 context EClass {
62     constraint UniqueEClass{
63
64         guard : self.isTypeOf(EClass)
65
66         check : EClass2NodeTraceLink.all.select(e|e.EClass.target = self).
           size() < 2
67
68         message : 'Multiple links of type EClass2Node found for EClass ' +
           self
69     }
70 }
71 context Node {
72
73     constraint UniqueNode{
74
75         guard : self.isTypeOf(Node)
76
77         check : EClass2NodeTraceLink.all.select(e|e.Node.target = self).
           size() < 2
78
79         message : 'Multiple links of type EClass2Node found for Node ' +
           self
80     }
81 }
82 context EClass {
83     constraint OneForEachEClass{
84
85         guard : self.isTypeOf(EClass)
86
87         check : EClass2DiagramLabelTraceLink.all.exists(e|e.EClass.target
           = self)
88
89         message : 'No links of type EClass2DiagramLabel found for EClass '
           + self

```

```

90  }
91  }
92  context DiagramLabel {
93    constraint OneForEachDiagramLabel{
94
95      guard : self.isTypeOf(DiagramLabel)
96
97      check : EClass2DiagramLabelTraceLink.all.exists(e|e.DiagramLabel.
          target = self)
98
99      message : 'No links of type EClass2DiagramLabel found for
          DiagramLabel ' + self
100  }
101  }
102  context EClass {
103    constraint UniqueEClass{
104
105      guard : self.isTypeOf(EClass)
106
107      check : EClass2DiagramLabelTraceLink.all.select(e|e.EClass.target
          = self).size() < 2
108
109      message : 'Multiple links of type EClass2DiagramLabel found for
          EClass ' + self
110  }
111  }
112  context DiagramLabel {
113    constraint UniqueDiagramLabel{
114
115      guard : self.isTypeOf(DiagramLabel)
116
117      check : EClass2DiagramLabelTraceLink.all.select(e|e.DiagramLabel.
          target = self).size() < 2
118
119      message : 'Multiple links of type EClass2DiagramLabel found for
          DiagramLabel ' + self
120  }
121  }
122  context EClass {
123    constraint OneForEachEClass{
124

```

```
125   guard : self.isTypeOf(EClass)
126
127   check : EClass2FigureDescriptorTraceLink.all.exists(e|e.EClass.
      target = self)
128   message : 'No links of type EClass2FigureDescriptor found for
      EClass ' + self
129 }
130 }
131 context FigureDescriptor {
132   constraint OneForEachFigureDescriptor{
133
134     guard : self.isTypeOf(FigureDescriptor)
135
136     check : EClass2FigureDescriptorTraceLink.all.exists(e|e.
      FigureDescriptor.target = self)
137     message : 'No links of type EClass2FigureDescriptor found for
      FigureDescriptor ' + self
138   }
139 }
140 context EClass {
141   constraint UniqueEClass{
142
143     guard : self.isTypeOf(EClass)
144
145     check : EClass2FigureDescriptorTraceLink.all.select(e|e.EClass.
      target = self).size() < 2
146
147     message : 'Multiple links of type EClass2FigureDescriptor found
      for EClass ' + self
148   }
149 }
150 context FigureDescriptor {
151   constraint UniqueFigureDescriptor{
152
153     guard : self.isTypeOf(FigureDescriptor)
154
155     check : EClass2FigureDescriptorTraceLink.all.select(e|e.
      FigureDescriptor.target = self).size() < 2
156
157     message : 'Multiple links of type EClass2FigureDescriptor found
      for FigureDescriptor ' + self
```

```

158 }
159 }
160 context EPackage {
161     constraint OneForEachEPackage{
162
163         guard : self.isTypeOf(EPackage)
164
165         check : EPackage2PaletteTraceLink.all.exists(e|e.EPackage.target =
            self)
166         message : 'No links of type EPackage2Palette found for EPackage '
            + self
167
168     }
169 }
170 context EPackage {
171     constraint UniqueEPackage{
172
173         guard : self.isTypeOf(EPackage)
174
175         check : EPackage2PaletteTraceLink.all.select(e|e.EPackage.target =
            self).size() < 2
176
177         message : 'Multiple links of type EPackage2Palette found for
            EPackage ' + self
178     }
179 }
180 context EClass {
181     constraint OneForEachEClass{
182
183         guard : self.isTypeOf(EClass)
184
185         check : EClass2CreationToolTraceLink.all.exists(e|e.EClass.target
            = self)
186         message : 'No links of type EClass2CreationTool found for EClass '
            + self
187     }
188 }
189 context CreationTool {
190     constraint OneForEachCreationTool{
191
192         guard : self.isTypeOf(CreationTool)

```

```

193
194     check : EClass2CreationToolTraceLink.all.exists(e|e.CreationTool.
        target = self)
195     message : 'No links of type EClass2CreationTool found for
        CreationTool ' + self
196 }
197 }
198 context EClass {
199     constraint UniqueEClass{
200
201         guard : self.isTypeOf(EClass)
202
203         check : EClass2CreationToolTraceLink.all.select(e|e.EClass.target
        = self).size() < 2
204
205         message : 'Multiple links of type EClass2CreationTool found for
        EClass ' + self
206     }
207 }
208 context CreationTool {
209     constraint UniqueCreationTool{
210
211         guard : self.isTypeOf(CreationTool)
212
213         check : EClass2CreationToolTraceLink.all.select(e|e.CreationTool.
        target = self).size() < 2
214
215         message : 'Multiple links of type EClass2CreationTool found for
        CreationTool ' + self
216     }
217 }

```

Listing E.2: EVL User-Defined Constraints for the Filesystem Case-Study

```

1 import "ECoreUtil.eol";
2
3 context EPackage2CanvasTraceLink {
4     //They should have the same name
5     constraint sameName {
6         check : self.EPackage.name.equals(self.Canvas.name)
7

```

```
8     message : "The name of the" + self.EPackage.name + "EPackage is not
      the same with the name of the" +self.Canvas.name+" Palette."
9   }
10 }
11
12 context EClass2ConnectionTraceLink {
13   //The annotation link end must be a gmf.link
14   constraint IsGmfLink {
15     check : self.EClass.isLink()
16
17     message : "EClass " + self.EClass.name + " is not annotated as
      @gmf.link"
18   }
19 }
20
21 context EClass2NodeTraceLink {
22   //The annotation link end must be a gmf.node
23   constraint IsGmfNode {
24     check : self.EClass.isNode()
25
26     message : "EClass " + self.EClass.name + " is not annotated as
      @gmf.node"
27   }
28 }
29
30 context EPackage2PaletteTraceLink {
31   //They should have the same name
32   constraint sameName {
33     check : (self.EPackage.name +'Palette').equals(self.Palette.~title
      )
34
35     message : "The name of the" + self.EPackage.name + "EPackage match
      the title of the" +self.Palette.~title
36   }
37 }
```

References

- SERENA SOFTWARE INC. (2010). Requirements traceability management (rtm) version 5.6. <http://www.serena.com/Products/rtm/home.asp>. 59
- AIZENBUD-RESHEF, N., PAIGE, R.F., RUBIN, J., SHAHAM-GAFNI, Y. & KOLOVOS, D.S. (2005). Operational semantics for traceability. In *Proceedings of the 1st Traceability Workshop, ECMDA-FA*. 51, 52, 92
- AIZENBUD-RESHEF, N., NOLAN, B., RUBIN, J. & SHAHAM-GAFNI, Y. (2006a). Model traceability. *IBM Systems Journal*, **45**. 1, 2, 5, 11, 12, 18, 20, 56, 61, 93, 106, 156, 211
- AIZENBUD-RESHEF, N., NOLAN, B.T., RUBIN, J. & SHAHAM-GAFINI, Y. (2006b). Model Traceability. *IBM Systems Journal*. 46
- ALANEN, L.J.P.I., M. & TRUSCAN, D. (2003). Realizing a model driven engineering process. Tech. rep., TUCS. 73, 74, 76, 77
- ALANEN, M. & PORRES, I. (2003). Difference and Union of Models. 144
- ALBINET, A., BOULANGER, J.L., DUBOIS, H., PERALDI-FRATI, M.A., SOREL, Y. & VAN, Q.D. (2007). Model-based methodology for requirements traceability in embedded systems. In *Proceedings of the 3rd Traceability Workshop, ECMDA-FA*. 15
- ALENCAR, F., CASTRO, J., CYSNEIROS, G. & MYLOPOULOS, J. (2000). From Early Requirements Modeled by i* Technique to Later Requirements Modeled in Precise UML. In *In Proc. of the III Workshop de Engenharia de Requisitos, Rio de Janeiro, Brasil*. 128, 132
- ALEXANDER, I. (2002). Towards automatic traceability in industrial practice. In *Proceedings of the 1st International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE'02)*. 63

- ALEXANDER, I. (2003). Semiautomatic tracing of requirement versions to use cases: Experiences and challenges. In *2nd International Workshop on Traceability in Emerging Forms of Software Engineering*. 30, 57
- ALPHAWORKS, I. (2005). Emfatic language for emf development. <http://www.alphaworks.ibm.com/tech/emfatic>. 193
- AMAR, B., LEBLANC, H. & COULETTE, B. (2008). A traceability engine dedicated to model transformation for software engineering. In *Proceedings of the 4th Traceability Workshop, ECMDA-FA*. viii, 40, 41, 83, 86, 94
- ANDERSON, K.M., SHERBA, S.A. & LEPHTIEN, W.V. (2002). Towards large-scale information integration. In *Proceedings of the 24th International Conference on Software Engineering, Orlando, FL, USA*. 32
- ANTONIOL, G., CANFORA, G., CASAZZA, G., LUCIA, A.D. & MERLO, E. (2002). Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering*, **28**, pp. 970 – 983. 20, 25, 26, 56, 83, 87, 92, 96
- ARKLEY, P. & RIDDLE, S. (2005). Overcoming the traceability benefit problem. In *Proceedings of the 13th IEEE International Conference on Requirements Engineering*. viii, 53, 54, 55, 56, 57, 63
- ASUNCION, H.U., FRANÇOIS, F. & TAYLOR, R.N. (2007). An end-to-end industrial software traceability tool. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. 63
- ASUNCION, H.U., ASUNCION, A.U. & TAYLOR, R.N. (2010). Software traceability with topic modeling. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE '10*, 95–104, ACM, New York, NY, USA. 136
- ATKINSON, C. & KHNE, T. (2001). The essence of multilevel metamodeling. In *Proceedings of the 4th International Conference on the UML 2001*. 73
- BARBERO, M., FABRO, M.D.D. & BÉZIVIN, J. (2007). Traceability and provenance issues in global model management. In *Proceedings of the 3rd Traceability Workshop, ECMDA-FA*. 41
- BAYERAND, J. & WIDEN, T. (2001). Introducing traceability to product lines. In *Lecture Notes In Computer Science; Revised Papers from the 4th International Workshop on Software Product-Family Engineering*. 21, 34, 83, 95

- BECK, K. (1999). *Extreme Programming Explained*. Addison-Wesley. 75
- BEHRENS, T. (2007). Never 'without a trace': Practical advice on implementing traceability. IBM Technical Library. 16
- BEZIVIN, J. (2005). On the unification power of models. *Journal on Software and Systems Modeling*, 171–188. 72
- BIANCHI, A., VISAGGIO, G. & FASOLINO, A.R. (2000). An exploratory case study of the maintenance effectiveness of traceability models. In *Proceedings of the 8th International Workshop on Program Comprehension*. 53
- BIANCHI, A., CAIVANO, D., LANUBILE, F. & VISAGGIO, G. (2001). Evaluating software degradation through entropy. In *Proceedings of the 11th International Software Metrics Symposium*. 48
- BOLDYREFF, C., NUTTER, D. & RANK, S. (2002). Active artefact management for distributed software engineering. In *Proceedings of the 26th International Computer Software and Applications Conference on Prolonging Software Life: Development and Redevelopment*. 18, 61
- BOWEN, .S.V., J.P. (1994). Formal methods: Epideictic or apodeictic? *Software Engineering Journal*. 71
- BROOKS, F.P., JR. (1975). The mythical man-month. In *Proceedings of the international conference on Reliable software*, 193–, ACM, New York, NY, USA. 174
- BROOKS, J., F. P. (1987). No silver bullet: Essence and accidents of software engineering. *IEEE Computer*, **20**, 177–203. 1
- CHAN, Z. & PAIGE, R. (2005). Designing a domain-specific contract language. In *Proc. European Conference on MDA*, LNCS 3748, Springer-Verlag. 42
- CHAPMAN, S. (2005). Simmetrics: An open source extensible library of similarity and distance metrics. <http://www.dcs.shef.ac.uk/sam/simmetrics.html>. 160, 166
- CHAUVEL, F. & FLEUREY, F. (2010). Kermeta Language Overview. <http://kermeta.org/>, <http://www.kermeta.org>. 114
- CHOI, S., CHA, S. & TAPPERT, C. (2010). A survey of binary similarity and distance measures. *Journal of Systemics, Cybernetics and Informatics*, **8**, pp. 4348. 86
- CHRISTIAN NENTWICH, LICIA CAPRA, WOLFGANG EMMERICH AND ANTHONY FINKELSTEIN (2002). xlinkit: A Consistency Checking and Smart Link Generation Service. *ACM Transactions on Internet Technology*, **2**, 151–185. 108

- CIANCONE, A., FILIERI, A. & MIRANDOLA, R. (2010). Mantra: Towards model transformation testing. *Quality of Information and Communications Technology, International Conference on the*, **0**, 97–105. 170
- CLELAND-HUANG, J. & SCHMELZER, D. (2003). Dynamic tracing non-functional requirements through design pattern recognition. In *Proceedings of the 2nd International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE 2003)*. 29
- CLELAND-HUANG, J., CHANG, C.K. & GE, Y. (2002a). Supporting event based traceability through high-level recognition of change events. In *26th Computer Software and Applications Conference, Oxford, UK*. 49, 50, 93
- CLELAND-HUANG, J., CHANG, C.K., SETHI, G., JAVVAJI, K., HU, H. & XIA, J. (2002b). Automating speculative queries through event-based requirements traceability. In *Proceedings of the IEEE Joint International Requirements Engineering Conference, Essen, Germany*. 29
- CLELAND-HUANG, J., CHANG, C.K. & CHRISTENSEN, M.J. (2003). Event-based traceability for managing evolutionary change. *IEEE Transactions on Software Engineering*, **29**, pp. 796–810. 22, 46, 83, 174
- CLELAND-HUANG, J., SETTIMI, R., DUAN, C. & ZOU, X. (2005). Utilizing supporting evidence to improve dynamic requirements traceability. In *Proceedings of the 13th IEEE International Conference on Requirements Engineering*. 25, 96
- COLLINS-SUSSMAN, B., FITZPATRICK, B.W. & PILATO, C.M. (2004). *Version Control with Subversion*. o'Reilly. 51
- CONSTANTOPOULOS, P., JARKE, M., MYLOPOULOS, J. & VASSILIOU, Y. (1995). The software information base: a server for reuse. *he International Journal on Very Large Data Bases*, **4**, pp 1–43. 56
- CORMACK, R.M. (1971). A review of classification. *Journal of the Royal Statistical Society. Series A (General)*, **134**, 321–367. 79
- COSTA, M. & DA SILVA, A.R. (2007). Rt-mdd framework a practical approach. In *Proceedings of the 3rd Traceability Workshop, ECMDA-FA*. viii, 21, 22, 83, 91, 94
- COWLING, A.J. (2005). The role of modelling in the software engineering curriculum. *The Journal of Systems & Software*. viii, 70
- CRAMÉR, H. (1946). *Mathematical Methods of Statistic*. Princeton: Princeton University Press. 92

- DAHLSTEDT, A.G. & PERSSON, A. (2003). Requirements interdependencies - moulding the state of research into a research agenda. In *Ninth International Workshop on Requirements Engineering: Foundation for Software Quality (REFSQ 2003), held in conjunction with CAiSE 2003*. viii, 39, 80
- DASHOFY, E.M., VAN DER HOEK, A. & TAYLOR, R.N. (2001). A highly-extensible, xml-based architecture description language. In *Working IEEE / IFIP Conference on Software Architecture, Amsterdam, The Netherlands*. 51
- DAVIS, H.C. (1998). Referential integrity of links in open hypermedia systems. In *Proceedings of the 9th ACM conference on Hypertext and hypermedia: links, objects, time and space, Pittsburgh, Pennsylvania, United States*, pp. 207–216. 48
- DAVIS, P.H. & HEYWOOD, V.H. (1963). *Principles of Angiosperm Taxonomy*. Princeton: Van Nostrand. 82
- DEERWESTER, S., DUMAIS, G.W., FURNAS, S.T., LANDAUER, T.K. & HARSHMAN, R. (1990). Indexing by latent semantic analysis. *Journal of the American Society for Information Science*, **41**, pp. 391–407. 24
- DHOOLIA, P., MANI, S., SINHA, V. & SINHA, S. (2010). Debugging model-transformation failures using dynamic tainting. In *ECOOP 2010 Object Oriented Programming*, vol. 6183 of *Lecture Notes in Computer Science*, 26–51, Springer Berlin, Heidelberg. 189
- DICK, J. (2005). Rich traceability. In *Proceedings of the 3rd international workshop on Traceability in emerging forms of software engineering*. 21, 35, 40
- DIJKSTRA, E.W. (1976). *A Discipline of Programming*. Prentice Hall, Inc. 1
- DMGES, R. & POHL, K. (1998). Adapting traceability environments to project-specific needs. *Communications of the ACM*, **41**, pp. 54–62. 65
- DOYLE, L. & BECKER, J. (1975). *Information Retrieval and Processing*. Melville Publishing Company, Los Angeles. 23
- DRIVALOS, N., R. F. PAIGE, K.J.F. & KOLOVOS, D.S. (2008). Towards rigorously defined model-to-model traceability. In *ECMDA Traceability WS*. 109
- DRIVALOS, N., KOLOVOS, D.S., PAIGE, R.F. & FERNANDES, K.J. (2009). Engineering a DSL for software traceability. In *1st International Conference on Software Language Engineering (SLE 2008), Revised Selected Papers, Springer-Verlag*, pp. 151–167. 189

- DRIVALOS, N., KOLOVOS, D.S., PAIGE, R. & FERNANDES, K. (2008). Engineering a Domain-Specific Language for Software Traceability. In *Proc. Software Language Engineering, Toulouse, France*. 109
- EADS, D. (2008). Hcluster: Hierarchical Clustering for SciPy. 89
- EBNER, G. & KAINDL, H. (2002). Tracing all around in reengineering. *IEEE Software*, **19**, pp. 70–77. 56
- ECLIPSE FOUNDATION (2010a). Eclipse Modelling Framework Project. <http://www.eclipse.org/modeling/emf>. 47, 71, 108, 111, 123, 164, 177
- ECLIPSE FOUNDATION (2010b). Graphical Modeling Framework. <http://www.eclipse.org/modeling/gmp/>. 123
- ECLIPSE FOUNDATION (2011a). Eclipse. <http://www.eclipse.org/>. 178
- ECLIPSE FOUNDATION (2011b). Graphical modeling project (gmp). <http://www.eclipse.org/modeling/gmp/?project=gmf-tooling>. ix, 190
- ECLIPSE FOUNDATION (2011c). Java development tools. <http://www.eclipse.org/jdt/>. 177
- ECMDA Traceability WS (2008). *Proceedings of the 3rd Traceability Workshop, ECMDA-FA*. 262, 276
- EFFTINGE, S. (2006a). oAW xText - A framework for textual DSLs. In *Workshop on Modeling Symposium at Eclipse Summit*. 123
- EFFTINGE, S. (2006b). openArchitectureWare 4.1 Xpand. Language reference. http://www.eclipse.org/gmt/oaw/doc/4.1/r20_xPandReference.pdf. 120
- EGYED, A. (2006). *Value-Based Software Engineering*, chap. Tailoring Software Traceability to Value-Based Needs. Springer Berlin Heidelberg. 66, 67
- EGYED, A. (2009). Value-based requirements traceability: Lessons learned. *Lecture Notes in Business Information Processing*, **14**, pp. 240–257. 66
- EGYED, A. & GRUNBACHER, P. (2002). Automating requirements traceability: Beyond the record & replay paradigm. In *Proceedings of the 17th IEEE International Conference on Automated Software Engineering*. 20, 83, 96
- EGYED, A. & GRUNBACHER, P. (2005). Supporting software understanding with automated requirements traceability. *International Journal of Software Engineering and Knowledge Engineering*, **15**, pp. 783–810. 31

- EGYED, A., BIFFL, S., HEINDL, M. & GRÜNBACHER, P. (2005). A value-based approach for understanding cost-benefit trade-offs during automated software traceability. In *Proceedings of the 3rd international workshop on Traceability in emerging forms of software engineering, TEFSE '05*, 2–7, ACM, New York, NY, USA. 98, 100, 138
- FABRO, M.D.D., BEZIVIN, J., JOUAULT, F., BRETON, E. & GUELTAS, G. (2005). AMW: A Generic Model Weaver. In *Proceedings of IDM05*. viii, 43, 44, 46, 83, 93, 95
- FALLERI, J., HUCHARD, M. & NEBUT, C. (2006). Towards a traceability framework for model transformations in kermeta. In *Proceedings of the 2nd Traceability Workshop, ECMDA-FA*. 33, 40, 83, 91, 94
- FIUTEM, R. & ANTONIOL, G. (1998). Identifying design-code inconsistencies in object-oriented software: a case study. In *Proceedings of the International Conference on Software Maintenance*. 54
- FLEUREY, F., BAUDRY, B., FRANCE, R. & GHOSH, S. (2007). A generic approach for automatic model composition. In *In Proc. AOM at MoDELS, 2007*. 144
- FLYNN, R. & DORFMAN, M. (1990). The automated requirements traceability system (ARTS): An experience of eight years. *System and Software Requirements Engineering*, pp. 423–438. 65
- FOR AERONAUTICS (RTCA), R.T.C. (1982). DO-178B: Software considerations in airborne systems and equipment certification. 54
- FREI, H.P. & STIEGER, D. (1995). The use of semantic links in hypertext information retrieval. *Inf. Process. Manage.*, **31**, 1–13. 96
- FRITZSCHE, M., JOHANNES, J., ZSCHALER, S., ZHEREBTSOV, A., & TEREKHOV, A. (2008). Application of tracing techniques in model-driven performance engineering. In *Proceedings of the 4th Traceability Workshop, ECMDA-FA*. 54
- G. KAPPEL, H.K.G.K.T.R.W.R.W.S., E. KAPSAMMER & WIMMER., M. (2006). On models and ontologies. In *A Layered Approach for Model-based Tool Integration*, 11–27. 68
- GERBER A., K.R.J.S.A.W., M. LAWLEY (2002). Transformation: The missing link of mda. In *Graph Transformation: First International Conference (ICGT 2002)*. 76
- GILLS, M. (2005). Survey of traceability models in IT projects. In *Proceedings of the 1st Traceability Workshop, ECMDA-FA*. 63

- GITZEL, R. & HILDENBRAND, T. (2005). A taxonomy of metamodel hierarchies. Tech. rep., Department of Information Systems, Universitt Mannheim. 73
- GITZEL, R. & KORTHAUS, A. (2004). The role of metamodeling in model-driven development. In *Proceedings of the 8th World Multi-Conference on Systemics, Cybernetics and Informatics (SCI2004)*. 72
- GOGUEN, J.A. (1996). Formality and informality in requirements engineering. In *Proceedings of the 2nd International Conference on Requirements Engineering (ICRE '96)*. 65
- GÖKNIL, A., KURTEV, I. & VAN DEN BERG, K. (2010). Tool support for generation and validation of traces between requirements and architecture. In *Proceedings of the 6th Traceability Workshop, ECMFA*. 46
- GORP, P.V. & JANSSENS, D. (2005). CAViT: a consistency maintenance framework based on visual model transformation and transformation contracts. In *J. Cordy, R. Limmel, and A. Winter, editors, Transformation Techniques in Software Engineering, number 05161 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum fr Informatik (IBFI), Schloss Dagstuhl, Germany*. 33
- GOTEL, O. & FINKELSTEIN, A. (1994). An analysis of the requirements traceability problem. In *Proceedings of the First International Conference on Requirements Engineering (RE'94)*, pp. 94–101. 11, 17, 18, 53, 57, 67
- GOTEL, O. & FINKELSTEIN, A. (1995). Contribution structures [requirements artifacts]. In *Proceedings of the Second IEEE International Symposium on Requirements Engineering*. 54
- GRAMMEL, B. & VOIGT, K. (2009). Foundations for a generic traceability framework in model-driven software engineering. In *Proceedings of the 5th Traceability Workshop, ECMDA-FA*. 41, 83, 91, 92, 94
- GRAU, G., CARES, C., FRANCH, X. & NAVARRETE, F.J. (2006). A comparative analysis of i* agent-oriented modelling techniques. In *18th International Conference on Software Engineering and Knowledge Engineering, SEKE 2006*, 657–663. 128
- GRECHANIK, M., MCKINLEY, K.S. & PERRY, D.E. (2007). Recovering and using use-case-diagram-to-source-code traceability links. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering*. 29, 83, 93, 96
- GROUP, O.M. (2011). Xml metadata interchange. <http://www.omg.org/spec/XMI/>. 178

- GUERRA, E., DE LARA, J., KOLOVOS, D.S. & PAIGE, R.F. (2010). Inter-modelling: from theory to practice. In *Proceedings of the 13th international conference on Model driven engineering languages and systems: Part I, MODELS'10*, 376–391, Springer-Verlag, Berlin, Heidelberg. 43
- GUSFIELD, D. (2007). *Algorithms on strings, trees, and sequences : computer science and computational biology*. Cambridge Univ. Press. 166
- HAEFNER, J. (2003). *Modeling biological systems. Principles and applications.*. Chapman and Hall. New York. 69
- HAPKE, M., JASZKIEWICZ, A., KOWALCZYKIEWICZ, K., WEISS, D. & ZIELNIEWICZ, P. (2004). Ophelia: Open platform for distributed software development. In *Open Source for an Information and Knowledge Society: Proceedings of the Open Source International Conference. Malaga, Spain*. 60
- HARROLD, M.J. (2000). Testing: a roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, 61–72, ACM, New York, NY, USA. 169
- HAYES, J.H. & DEKHTYAR, A. (2005). Humans in the traceability loop: can't live with 'em, can't live without 'em. In *Proceedings of the 3rd international workshop on Traceability in emerging forms of software engineering*. 67
- HAYES, J.H., DEKHTYAR, A. & OSBORNE, J. (2003). Improving requirements tracing via information retrieval. In *Proceedings of the 11th IEEE International Conference on Requirements Engineering*. 25, 83, 92, 96
- HAYES, J.H., DEKHTYAR, A., SUNDARAM, S.K. & HOWARD, S. (2004). Helping analysts trace requirements: An objective look. In *Proceedings of the Requirements Engineering Conference, 12th IEEE International*. 25, 83, 96
- HAYES, J.H., DEKHTYAR, A. & SUNDARAM, S.K. (2006). Advancing candidate link generation for requirements tracing: The study of methods. *IEEE Transactions on Software Engineering*, **32**, pp. 4–19. 25, 83, 96
- HEAVEN, W. & FINKELSTEIN, A. (2004). A UML Profile to Support Requirements Engineering with KAOS. *IEEE Proceedings: Software*. 47
- HEINDL, M. & BIFFL, S. (2005). A case study on value-based requirements tracing. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*. 66

- HEINDL, M. & BIFFL, S. (2008). Modeling of requirements tracing. *Lecture Notes In Computer Science, Balancing Agility and Formalism in Software Engineering: Second IFIP TC 2 Central and East European Conference on Software Engineering Techniques, CEE-SET 2007, Poznan, Poland, Revised Selected Papers*, pp. 267–278. 66
- HELMING, J., KOEGEL, M., NAUGHTON, H., DAVID, J. & SHTEREV, A. (2009). Traceability-based change awareness. In *Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems, MODELS '09*, 372–376, Springer-Verlag, Berlin, Heidelberg. 174
- HEMPEL, C.G. (1952). *Symposium: Problems of concept and theory formation in the social sciences, language and human rights*. Philadelphia: University of Pennsylvania Press. 79
- HERMAN, I., MELANÇON, G. & MARSHALL, M.S. (2000). Graph visualization and navigation in information visualization: A survey. *IEEE Transactions on Visualization and Computer Graphics*, **6**, pp. 24–43. 47
- HOFSTADTER, D. (1979). *Gdel, Escher, Bach*. Random House, New York. 69
- HOLAGENT (2005). Holagent corporation product rdd-100. <http://www.holagent.com/products/product1.html>. 58
- IBM (2010). Rational doors. <http://www-01.ibm.com/software/awdtools/doors/productline/>. 21, 26, 30, 59
- IEEE (1990). IEEE Standard Glossary of Software Engineering Terminology. IEEE Std. 610.12-1990. 12
- ISO/IEC (1990). Information technology information resource dictionary systems (irds) framework. 60
- JING, Y.L. (2007). *A Testing Framework for Model Transformations*, 219–236. Springer. 169
- JIRAPANTHONG, W. & ZISMAN, A. (2007). Xtraque: traceability for product line systems. *Software and Systems Modeling*, **8**, pp.117–144. 28, 83, 96
- JOUAULT, F. (2005). Loosely coupled traceability for ATL. In *Proceedings of the 1st Traceability Workshop, ECMDA=-FA*. viii, 41, 42, 83, 91, 94, 107, 140
- JOUAULT, F. & BÉZIVIN, J. (2006). Km3: a dsl for metamodel specification. In *Proceedings of 8th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems, LNCS 4037, Bologna, Italy*. 43

- JOUAULT, F., ALLILAIRE, F., BÉZIVIN, J., KURTEV, I. & VALDURIEZ, P. (2006). ATL: a QVT-like transformation language. In *OOPSLA Companion*, 719–720. 32, 114, 146
- JOUAULT, F., BÉZIVIN, J. & BARBERO, M. (2009). Towards an advanced model-driven engineering toolbox. *Innovations in Systems and Software Engineering*, **5**, 5–12. viii, 97, 99
- KENT, S. (2003). Model driven engineering. In *Integrated Formal Methods: Third International Conference, IFM 2002, Turku, Finland*,. 2, 73, 75
- KIROVA, V., KIRBY, N., KOTHARI, D. & CHILDRESS, G. (2008). Effective requirements traceability: Models, tools, and practices. *Bell Labs Technical Journal*, **12**, pp. 143–157. 63
- KLEPPE, A. (2007). A Language Description is More than a Metamodel. In *Proc. 4th International Workshop on Software Language Engineering*, Nashville, USA. 114
- KOLOVOS, D. (2008a). *An Extensible Platform for Specification of Integrated Languages for Model Management*. Ph.D. thesis, Department of Computer Science, University of York. 139
- KOLOVOS, D. (2008b). *An Extensible Platform for Specification of Integrated Languages for Model Management*. Ph.D. thesis, Department of Computer Science, University of York. 139, 178
- KOLOVOS, D. (2009). Establishing correspondences between models with the epsilon comparison language. In *Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications, ECMDA-FA '09*, Springer-Verlag, Berlin, Heidelberg. 139, 144, 181
- KOLOVOS, D.S. (2007). Editing emf models with exceed (extended emf editor). Tech. rep., Department of Computer Science, University of York. 123, 182
- KOLOVOS, D.S. & PAIGE, R.F. (2010). Extensible Platform for Specification of Integrated Languages for mOdel maNagement (Epsilon). <http://www.eclipse.org/gmt/epsilon>. 32, 83, 94, 143, 177
- KOLOVOS, D.S., PAIGE, R.F. & POLACK, F. (2006a). The epsilon object language. In *Proceedings of the European Conference in Model Driven Architecture (EC-MDA) 2006, Bilbao, Spain*. 180
- KOLOVOS, D.S., PAIGE, R.F. & POLACK, F. (2006b). On Demand Merging of Traceability Links with Models. In *ECMDA - TW: Traceability Workshop, at European Conference on Model Driven Architecture, Bilbao, Spain*. 46, 47

- KOLOVOS, D.S., PAIGE, R.F. & POLACK, F. (2006c). On-demand merging of traceability links with models. In *Proceedings of the 2nd Traceability Workshop, ECMDA-FA*. 107
- KOLOVOS, D.S., PAIGE, R.F., POLACK, F.A.C. & ROSE, L.M. (2007). Update transformations in the small with the epsilon wizard language. *Journal of Object Technology (JOT), Special Issue for TOOLS Europe*. ix, 174, 175, 179, 180, 182
- KOLOVOS, D.S., PAIGE, R.F. & POLACK, F.A. (2008). The Epsilon Transformation Language. In *Proc. 1st International Conference on Model Transformation, Zurich, Switzerland*, to appear. 114, 181
- KOLOVOS, D.S., ROSE, L.M., PAIGE, R.F. & POLACK, F.A.C. (2009). Raising the level of abstraction in the development of gmf-based graphical model editors. In *Proceedings of the 2009 ICSE Workshop on Modeling in Software Engineering, MISE '09*, 13–19, IEEE Computer Society, Washington, DC, USA. 189, 191, 192, 193, 194
- KOLOVOS, D.S., ROSE, L.M., ABID, S.B., PAIGE, R.F., POLACK, F.A.C. & BOTTERWECK, G. (2010). Taming emf and gmf using model transformation. In *Proceedings of the 13th international conference on Model driven engineering languages and systems: Part I, MODELS'10*, 211–225, Springer-Verlag, Berlin, Heidelberg. ix, 191, 192, 196
- KOLOVOS, D.S., PAIGE, R.F. & POLACK, F.A. (2007). On the Evolution of OCL for Capturing Structural Constraints in Modelling Languages. In *Proc. Dagstuhl Workshop on Rigorous Methods for Software Construction and Analysis*. 108, 114, 181
- KUROPKA, D. (2004). Modelle zur Repräsentation Natürlichsprachlicher Dokumente - Information-Filtering und -Retrieval mit relationalen Datenbanken. *Logos Verlag*. viii, 24
- KURTEV, I., DEE, M., GOKNIL, A. & VAN DEN BERG, K. (2007). Traceability-based change management in operational mappings. In *Proc. Workshop on Traceability*. 33
- LEE, C., GUADAGNO, L. & JIA, X. (2003). An agile approach to capturing requirements traceability. In *Proceedings of the 2nd International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE 2003)*,. 20
- LESEURE, M. (2000). Manufacturing strategies in the hand tool industry. *International Journal of Operations & Production Management*, **20**, pp. 1475–1487. 80
- LIMON, A. & GARBAJOSA, J. (2005). The need for a unifying traceability scheme. In *Proceedings of the 1st Traceability Workshop, ECMDA-FA*. 42

- LIN, J., LIN, C.C., CLELAND-HUANG, J., SETTIMI, R., AMAYA, J., BEDFORD, G., BERENBACH, B., KHADRA, O.B., DUAN, C. & ZOU, X. (2006). Poirot: A distributed tool supporting enterprise-wide automated traceability. In *14th IEEE International Requirements Engineering Conference (RE'06)*. 26, 83, 96
- LINDVALL, M. & SANDAHL, K. (1996). Practical implications of traceability. *Software Practice and Experience*, **26**, pp. 1161–1180. 18, 19
- LINDVALL, M. & SANDAHL, K. (1998). Traceability aspects of impact analysis in object-oriented systems. *Journal of Software Maintenance: Research and Practice*, **10**, pp. 37–57. 63
- LORMANS, M. & VAN DEURSEN, A. (2005). Reconstructing requirements coverage views from design and test using traceability recovery via lsi. In *Proceedings of the 3rd international workshop on Traceability in emerging forms of software engineering*. 26
- LORMANS, M. & VAN DEURSEN, A. (2006). Can lsi help reconstructing requirements traceability in design and test? In *Proceedings of the Conference on Software Maintenance and Reengineering*. 26
- LUCIA, A.D., OLIVETO, R. & TORTORA, G. (2004). ADAMS: An artifact-based process support system. In *Proceedings of 16th International Conference on Software Engineering and Knowledge Engineering (Banff, Alberta, Canada, June)*. F. Maurer and G. Ruhe, Eds. 26
- LUCIA, A.D., FASANO, F., OLIVETO, R. & TORTORA, G. (2007). Recovering traceability links in software artifact management systems using information retrieval methods. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, **16**. 26, 83
- LUCIA, A.D., OLIVETO, R. & TORTORA, G. (2008). Adams re-trace: traceability link recovery via latent semantic indexing. In *Proceedings of the 30th international conference on Software engineering*. 26, 83, 96
- LUDEWIG, J. (2003). Models in software engineering an introduction. *Software and Systems Modeling*, **2**, 5–14. 68, 69, 70, 71
- MÄDER, P. (2008). Tracemaintainer - automated traceability maintenance. In *16th IEEE International Requirements Engineering Conference, Barcelona, Catalunya, Spain*. 83, 94, 95
- MÄDER, P., GOTEL, O. & PHILIPPOW, I. (2008). Rule-based maintenance of post-requirements traceability relations. In *Proceedings of the 2008 16th IEEE International Requirements Engineering Conference*. 49, 157

- MÄDER, P., GOTEL, O. & PHILIPPOW, I. (2009). Enabling automated traceability maintenance through the upkeep of traceability relations. In *Proceedings of the 5th European Conference on Model-Driven Architecture, Foundations and Applications, Enschede, the Netherlands*. 51, 157
- MAEDER, P. & RIEBISCH, M. (2007). Customizing traceability links for the unified process. In *Software Architectures, Components, and Applications. QoSA 2007 - Revised Selected Papers. Springer LNCS*. 19
- MALETIC, J., COLLARD, M.L. & SIMOES, B. (2005). An xml-based approach to support the evolution of model-to-model traceability links. In *Proceedings of the 3rd International Workshop on Traceability in Emerging Forms of Software Engineering*. 52, 83, 95
- MALETIC, J.I. & MARCUS, A. (2001). Supporting program comprehension using semantic and structural information. In *Proceedings of the 23rd International Conference on Software Engineering*. 56
- MALETIC, J.I., MUNSON, E.V., MARCUS, A. & NGUYEN., T.N. (2003). Using a hypertext model for traceability link conformance analysis. In *Proceedings of the 2nd International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE)*. 26, 59, 83, 96
- MARCUS, A. & MALETIC, J.I. (2003). Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Proceedings of the 25th International Conference on Software Engineering*. 20, 56, 83, 96
- MARCUS, A., XIE, X. & POSHYVANYK, D. (2005). When and how to visualize traceability links? In *Proceedings of the 3rd international workshop on Traceability in emerging forms of software engineering*. 45
- MCCARTHY, I.P. & TSINOPOULOS, C. (2003). Strategies for agility: an evolutionary and configurational approach. *Strategies for agility: an evolutionary and configurational approach*, **14**, pp. 103–113. 80, 84
- MENS, T., CZARNECKI, K. & GORP, P.V. (2005). A taxonomy of model transformations. In *Dagstuhl 04101 Language Engineering for Model-Driven Software Development*. 75, 138, 173
- MILLER, G.A. (1995). Wordnet: A lexical database for english. *Communications of the ACM*, **38**, 39–41. 160
- MOORE, B. (2004). *Eclipse development using the graphical editing framework and the eclipse modeling framework*. IBM Corp., Riverton, NJ, USA. 190

- MUNSON, E.V. & NGUYEN, T.N. (2005). Concordance, conformance, versions, and traceability. In *Proceedings of the 3rd international workshop on Traceability in emerging forms of software engineering*. 40
- MURPHY, G.C., NOTKIN, D. & SULLIVAN, K. (1995). Software reflexion models: Bridging the gap between source and high-level models. In *Proceedings of the 3rd ACM SIGSOFT Symposium on Foundations of Software Engineering*. 20
- MURTA, L.G.P., VAN DER HOEK, A. & WERNER, C.M.L. (2006). Archtrace: Policy-based support for managing evolving architecture-to-implementation traceability links. In *21st IEEE/ACM International Conference on Automated Software Engineering, Tokyo, Japan*. 51, 83, 95
- MYLOPOULOS, J., BORGIDA, A., JARKE, M. & KOUBARAKIS, M. (1990). Telos: Representing knowledge about information systems. *ACM Transactions on Information Systems*, **8**, pp. 325–362. 31
- NATT OCH DAG, J., REGNELL, B., CARLSHAMRE, P., ANDERSSON, M. & KARLSSON, J. (2002). A feasibility study of automated natural language requirements analysis in market-driven development. *Requirements Engineering*. 27
- NATT OCH DAG, J., GERVASI, V., BRINKKEMPER, S. & REGNELL, B. (2005). A linguistic-engineering approach to large-scale requirements management. *IEEE Software*, **22**, pp. 32–39. 27, 83, 96
- NEAL, R.W.M. (1994). Why and how of requirements tracing. *IEEE Software*, **11**, pp.104 – 106. 53
- NETBEANS (2003). Netbeans metadata repository introduction (mdr and related standards). netbeans-uml-extender-plugin.googlecode.com/files/MDR-whitepaper.pdf. 180
- NGUYEN, T.N. & MUNSON, E.V. (2003). The software concordance: a new software document management environment. In *Proceedings of the 21st Annual International Conference on Documentation, San Francisco, CA, USA*. 58
- OBEO07 (2007). Acceleo Pro Traceability. http://www.acceleo.org/pages/additionnal_products/en, Last accessed June 2007. 33
- OBJECT MANAGEMENT GROUP (2002). Revised submission for MOF 2.0 Query/Views/Transformations RFP (ad/2002-04-10). <http://www.omg.org/cgi-bin/doc?ptc/03-10-04>. 114
- OBJECT MANAGEMENT GROUP (2003). UML 2.0 OCL specification. OMG Document, uRL <http://www.omg.org/cgi-bin/doc?ptc/03-10-14>. 108, 180

- OBJECT MANAGEMENT GROUP (2004). UML 2.0 infrastructure specification. OMG Document, uRL <http://www.omg.org/cgi-bin/doc?ptc/04-10-14>. 128, 130
- OBJECT MANAGEMENT GROUP (2006a). Meta Object Facility (MOF) 2.0 Core Specification. <http://www.omg.org/cgi-bin/doc?ptc/03-10-04>. 47, 71
- OBJECT MANAGEMENT GROUP (2006b). MOF models to text transformation language; final adopted specification. OMG Document 08-01-16, <http://www.omg.org/spec/MOFM2T/1.0/PDF>. 33
- OBJECT MANAGEMENT GROUP (2011a). Metaobject facility. <http://www.omg.org/mof/>. 111
- OBJECT MANAGEMENT GROUP (2011b). Model Driven Architecture. <http://www.omg.org/mda>. 12
- OBJECT MANAGEMENT GROUP (2011c). MOF QVT draft specification. OMG Document ptc/2007/07/07, <http://www.omg.org/cgibin/doc?ptc/2007-07-07>. 32
- OLDEVIK, J. (2011). MOFScript User Guide. <http://www.eclipse.org/gmt/mofscript/doc/MOFScript-User-Guide.pdf>. 120
- OLDEVIK, J. & AAGEDAL, J. (2005). Future Research Topics Discussion. Traceability Workshop, EC-MDA, <http://www.sintef.no/upload/10558/Future-Research-Topics.pdf>. 46
- OLDEVIK, J., NEPLE, T., GRONMO, R., AAGEDAL, J. & BERRE, A. (2005). Toward standardised model-to-text transformations. In *Proceedings of the First European Conference on Model Driven Architecture: Foundations and Applications, Nürnberg, Germany*, no. 3748 in LNCS, Springer. 33
- OLSEN, G. & OLDEVIK, J. (2007). Scenarios of traceability in model-to-text transformations. In *Proc. European Conference on MDA*, LNCS, Springer-Verlag. 32, 41, 42, 83, 91, 92, 94
- ORACLE (2011). Java programming language. <http://www.java.com/en/>. 177
- ØSTERBYE, K. & WIIL, U.K. (1996). The flag taxonomy of open hypermedia systems. In *Proceeding of the Seventh ACM Conference on Hypertext, Washington, DC, USA*. 32
- O'SULLIVAN, L.C.B.Y.L.L. (2003). Impact analysis and change management of uml models. In *Proceedings of the International Conference on Software Maintenance*. 54

- PAIGE, R., OLSEN, G., KOLOVOS, D., ZSCHALER, S. & POWER, C. (2008). Building model-driven engineering traceability classifications. In *Proceedings of the 4th Traceability Workshop, ECMDA-FA*, SINTEF Technical Report. 2, 12, 18, 42, 92, 98, 106, 107, 211, 215
- PARNAS, D.L. (1994). Software aging. In *Proceedings of the 16th international conference on Software engineering, ICSE '94*, 279–287, IEEE Computer Society Press, Los Alamitos, CA, USA. 174
- PIERCE, R.A. (1978). A requirements tracing tool. In *Proceedings of the software quality assurance workshop on Functional and performance issues*. 67, 96
- PILATO, M. (2004). *Version Control With Subversion*. O'Reilly & Associates, Inc., Sebastopol, CA, USA. 137
- PINHEIRO, F.A.C. (2004). *Perspectives on Software Requirements*, chap. Requirements Traceability, pp. 91–113. Kluwer Academic Publishers, The Netherlands. 11, 19, 20, 46, 80, 95, 149
- PINHEIRO, F.A.C. & GOGUEN, J.A. (1996). An object-oriented tool for tracing requirements. *IEEE Software*, **13**, pp. 52–64. 21, 29, 35, 46, 60, 83
- POHL, K. (1996a). Pro-art: Enabling requirements pre-traceability. In *Proceedings of the 2nd International Conference on Requirements Engineering (ICRE '96)*. 21, 31, 36, 46, 56, 59, 83, 96
- POHL, K. (1996b). *Process-Centered Requirements Engineering*. John Wiley & Sons, Inc. New York, NY, USA. 37
- POHL, M.J.K. (1992). Information systems quality and quality information systems. In *Proceedings of the IFIP WG8.2 Working Conference on The Impact of Computer Supported Technologies in Information Systems Development*. 57
- POHL, P.H.K., WEIDENHAUPT, K. & JARKE, M. (1999). Improving reviews by extended traceability. In *Proceedings of the Thirty-Second Annual Hawaii International Conference on System Sciences*. 56
- POWER, C., PETRIE, H., SWALLOW, D. & PAIGE, R.F. (2009). Pre-requirements traceability in universally accessible e-learning systems. In *under review*. 42
- QUEIROZ, K.D. & GOOD, D.A. (1997). Phenetic clustering in biology: A critique. *The Quarterly Review of Biology*, **72**, pp. 3–30. 89

- RAMESH, B. & DHAR, V. (1992). Supporting systems development by capturing deliberations during requirements engineering. *IEEE Transactions on Software Engineering*, **18**, pp. 498–510. 56
- RAMESH, B. & EDWARDS, M. (1993). Issues in the development of a requirements traceability model. In *Proceedings of the IEEE International Symposium on Requirements Engineering, San Diego, California*. 15, 53, 54, 56, 57
- RAMESH, B. & JARKE, M. (2001). Toward reference models of requirements traceability. *IEEE Trans. Software Eng.*, **27**, 58–93. viii, 11, 34, 35, 36, 37, 42, 83, 92, 95
- RAMESH, B., HARRINGTON, G., RONDEAU, K. & EDWARDS., M. (1993). A model of requirements traceability to support systems development. technical report nps-sm-93-017. Tech. rep., Naval Surface Warfare Center Dahlgren Division, 10901 New Hampshire Avenue, Silver Spring, Maryland 20903-5000, U.S.A. 13, 15
- RAMESH, K.M.B. (2007). Traceability-based knowledge integration in group decision and negotiation activities. *Decision Support Systems*, **43**, pp. 968–989. 46
- RIDLEY, M. (1993). *Evolution*. Blackwell Scientific Publications, Oxford. 80
- ROMESBURG, C.H. (1984). *Cluster Analysis for Researchers*. Belmont, CA: Lifetime Learning Publications. 82, 86, 87, 88, 89
- ROSE, L.M., PAIGE, R.F., KOLOVOS, D.S. & POLACK, F. (2008). The Epsilon Generation Language. In *Fourth European Conference on Model Driven Architecture: Foundations and Applications, Fraunhofer FOKUS, Berlin, Germany*, 1–16. 33, 83, 94, 120, 181
- ROSE, L.M., KOLOVOS, D.S., DRIVALOS, N., WILLIAMS, J.R., PAIGE, R.F., POLACK, F.A. & FERNANDES, K.J. (2010). Concordance: An efficient framework for managing model integrity. In *Proc. 6th European Conference on Modelling Foundations and Applications (ECMFA), June 2010, Paris, France (to appear)*. 163
- ROTHENBERG, J. (1989). *The Nature of Modeling in Artificial Intelligence, Simulation, and Modeling*. John Wiley and Sons, Inc. 69
- RUMMLER, A., GRAMMEL, B. & POHL, C. (2007). Improving traceability in model-driven development of business applications. In *Proceedings of the 3rd Traceability Workshop, ECMDA-FA*. 42
- SABETZADEH, M. & EASTERBROOK, S. (2005). Traceability in viewpoint merging: a model management perspective. In *Proceedings of the 3rd international workshop on Traceability in emerging forms of software engineering*. 56

- SANDAHL, M.L.K. (1996). Practical implications of traceability. In *SoftwarePractice & Experience*. 53
- SCHWARZ, H., EBERT, J. & WINTER, A. (2009). Graph-based traceability: a comprehensive approach. *Software and Systems Modeling*. 83, 91, 92, 94, 189
- SEIBEL, A., NEUMANN, S. & GIESE, H. (2010). Dynamic hierarchical mega models: comprehensive traceability and its efficient maintenance. *Software and Systems Modeling*, **9**, 493–528, 10.1007/s10270-009-0146-z. 189
- SEIDEWITZ, E. (2003). What models mean. *IEEE Software*. 69, 70, 71, 72
- SENDALL, S. & KOZACZYNSKI, W. (2003). Model transformation: The heart and soul of model-driven software development. *IEEE Software*, **20**, 42–45. 173
- SHARIF, B. & MALETIC, J.I. (2007). Using fine-grained differencing to evolve traceability links. International Symposium on Grand Challenges in Traceability (GCT'07). 52, 83, 95
- SHERBA, S.A., ANDERSON, K.M. & FAISAL, M. (2003). A framework for mapping traceability relationships. In *8th IEEE International Conference on Automated Software Engineering, Montreal, Quebec, Canada*. viii, 32, 45, 61, 62, 83, 93, 96
- SMITH, M., WEISS, D., WILCOX, P. & DEWER, R. (2003). The ophelia traceability layer. *Cooperative Methods and Tools for Distributed Software Processes*, pp. 150–161. 60
- SNEATH, P. & SOKAL, R.R. (1963). *Principles of Numerical Taxonomy*. W.H. Freeman, San Francisco. 80, 84
- SNEATH, P.H. & SOKAL, R.R. (1973). *Numerical Taxonomy: The Principles and Practice of Numerical Classification*. W.H. Freeman, San Francisco. 80, 81, 84
- SOFTEAM (2010). Modelio. <http://www.modeliosoft.com/>. 58
- SOMMERVILLE, I. (2004). *Software Engineering*. Pearson Education Limited, Essex England. 71
- SOMMERVILLE, I., SOMMERVILLE, I., SAWYER, P. & SAWYER, P. (1997). Viewpoints: principles, problems and a practical approach to requirements engineering. *Annals of Software Engineering*, **3**, 101–130. 135
- SOUSA, A., KULESZA, U., RUMMLER, A., ANQUETIL, N., MITSCHKE, R., MORAIRA, A., AMARAL, V. & ARAÚJO, J. (2008). A model-driven traceability framework to software product line development. In *ECMDA Traceability WS*. 41, 83, 95

- SPANOUDAKIS, G. & KIM, H. (2004). Supporting the reconciliation of models of object behaviour. *Software and Systems Modeling*, **3**, pp. 273–293. 54
- SPANOUDAKIS, G. & ZISMAN, A. (2005). Software traceability: a roadmap. *Handbook of Software Engineering and Knowledge Engineering*, **3**. 2, 12, 19, 20, 21, 35, 37, 38, 53, 56, 80, 83, 95, 149
- SPANOUDAKIS, G., D'AVILA GARCEZ, A.S. & ZISMA, A. (2003). Revising rules to capture requirements traceability relations: A machine learning approach. In *Proceedings of the Fifteenth International Conference on Software Engineering & Knowledge Engineering (SEKE'2003), Hotel Sofitel, San Francisco Bay, CA, USA*. 28, 83, 96
- SPANOUDAKIS, G., ZISMAN, A., PEREZ-MINANA, E. & KRAUSE, P. (2004). Rule-based generation of requirements traceability relations. *The Journal of Systems and Software*, pp. 105–127. 27, 28, 83, 96
- SPIVEY, J. (1989). *The Z Notation: A Reference Manual*.. Prentice Hall. 70
- STACHOWIAK, H. (1973). *Allgemeine Modeltheorie*. Springer Verlag. 71
- STATE, A.M.K. & MALETIC, J.I. (2003). Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Proceedings of the 25th International Conference on Software Engineering*. 26
- STEVENS, P. (2003). Small-scale xmi programming: a revolution in uml tool use? *Automated Software Engineering*. 76
- STONE, A. & SAWYER, P. (2005). Finding tacit knowledge by solving the pre-requirements tracing problem. In *Proceedings of the 11th International Workshop on Requirements Engineering : Foundation for Software Quality (REFSQ'05), Porto, Portugal*. 64
- TRATT, L. (2008). A change propagating model transformation language. *Journal of Object Technology*, **7**, 107–126. 174
- TREUDE, C., BERLIK, S., WENZEL, S. & KELTER, U. (2007). Difference computation of large models. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, ESEC-FSE '07, 295–304, ACM, New York, NY, USA*. 143, 144
- VAN DEN BERG, K., CONEJERO, J.M. & HERNNDEZ, J. (2006). Analysis of crosscutting across software development phases based on traceability. In *Proceedings of the 2006 international workshop on Early aspects at ICSE*. 56

- VANHOOFF, B., AYED, D., BAELEN, S.V., JOOSEN, W. & BERBERS, Y. (2007a). UniTI : A unified transformation infrastructure. In *G. Engels, B. Opdyke, D. C. Schmidt, F.Weil (eds), MoDELS, vol. 4735 of Lecture Notes in Computer Science, Springer*. 33, 83, 94
- VANHOOFF, B., BAELEN, S.V., JOOSEN, W. & BERBERS, E. (2007b). Traceability as input for model transformations. In *Proceedings of the 3rd Traceability Workshop, ECMDA-FA*. 41
- VON KNETHEN, A. & GRUND, M. (2003). Quatrace: A tool environment for (semi-) automatic impact analysis based on traces. In *Proceedings of the International Conference on Software Maintenance*. 54, 83, 96
- VON KNETHEN, A. & PAECH, B. (2002). A survey on tracing approaches in theory and practice. technical report 095.01/e. Tech. rep., Fraunhofer IESE. 2, 19, 39, 53, 80
- VON KNETHEN, A., PAECH, B. & HOUDEK, F.K.F. (2002). Systematic requirements recycling through abstraction and traceability. In *Proceedings of the 10th Anniversary IEEE Joint International Conference on Requirements Engineering*. 54, 55, 57
- W3C (2001). Xml linking language (xlink) version 1.0. <http://www.w3.org/TR/xlink/>. 28
- W3C (2007). Xquery 1.0: An xml query language. <http://www.w3.org/TR/xquery/>. 28
- WALDERHAUG, S., JOHANSEN, U., STAV, E. & AAGEDAL, J. (2006). Towards a generic solution for traceability in MDD. In *Proceedings of the 2nd Traceability Workshop, ECMDA-FA*. 40, 42, 83, 95
- WALDERHAUG, S., STAV, E., JOHANSEN, U. & OLSEN, G.K. (2008). *Designin Software-Intensive Systems: Methods and Principles*, chap. Traceability in Model-Driven Software Development, pp. 133–159. IGI Global. 2, 5, 92, 106, 211
- WENZEL, S., HUTTER, H. & KELTER, U. (2007). Tracing model elements. In *International Conference on Software Maintenance*, 104–113. 31, 83
- WHITEHEAD, E. (2000). *An analysis of the hypertext versioning domain*. Ph.D. thesis, University of California, Irvine. 58
- WIENANDS, C. & GOLM, M. (2009). Anatomy of a visual domain-specific language project in an industrial context. In *Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems, MODELS '09*, 453–467, Springer-Verlag, Berlin, Heidelberg. 191

- WIERINGA, R. (1995). An introduction to requirements traceability. Tech. rep., Faculty of Mathematics and Computer Science, University of Vrije, Amsterdam. 17, 45, 47, 65, 80
- WILLIAMS, J.R. & POLACK, F.A. (2010). Automated formalisation for verification of diagrammatic models. *Electronic Notes in Theoretical Computer Science*, **263**, 211 – 226, proceedings of the 6th International Workshop on Formal Aspects of Component Software (FACS 2009). 189
- WINKLER, S. & VON PILGRIM, J. (2009). A survey of traceability in requirements engineering and model-driven development. *Software and Systems Modeling*. 11, 20, 22, 64, 80, 135, 189
- WOODCOCK, J., J. & DAVIES (1996). *Using Z: Specification, Refinement, and Proof*. Prentice Hall. 71
- WORDSWORTH, J. (1996). *Software Engineering with B*. Addison Wesley Longman. 71
- YING, A.T.T., MURPHY, G.C., NG, R. & CHU-CARROLL, M.C. (2004). Predicting source code changes by mining change history. *IEEE Transactions on Software Engineering*, **30**, pp. 574–586. 30, 83, 96
- YU, E. (1997). Towards Modeling and Reasoning Support for Early-Phase Requirements Engineering. *Proc. RE-97 - 3rd International Symposium on Requirements Engineering, Annapolis*. 127, 128
- ZIMMERMANN, T., WEISGERBER, P., DIEHL, S. & ZELLER, A. (2004). Mining version histories to guide software changes. In *Proceedings of the 26th International Conference on Software Engineering*. 30, 83, 96
- ZISMAN, A., SPANOUDAKIS, G., PEREZ-MINANA, E. & KRAUSE, P. (2003). Tracing software requirements artifacts. In *Proceedings of International Conference on Software Engineering Research and Practice (Las Vegas, NV)*. 20, 83, 96
- ZOU, X., SETTIMI, R. & CLELAND-HUANG, J. (2006). Phrasing in dynamic requirements trace retrieval. In *Proceedings of the 30th Annual International Computer Software and Applications Conference*. 26, 83, 96
- ZSCHALER, S., KOLOVOS, D.S., DRIVALOS, N., PAIGE, R.F. & RASHID, A. (2009). Domain-specific metamodelling languages for software language engineering. In *Software Language Engineering, LNCS*. Springer, Berlin. 104