View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Repositorio da Producao Cientifica e Intelectual da Unicamp

Universidade Estadual de Campinas
Faculdade de Engenharia Elétrica e de Computação

Bruno Luis Pires de Azevedo

A Proposal for Traceability in Software Development

Uma Proposta para Rastreabilidade no Desenvolvimento de Software

CAMPINAS
2019

Bruno Luis Pires de Azevedo

A Proposal for Traceability in Software Development

Uma Proposta para Rastreabilidade no Desenvolvimento de Software

Dissertation presented to the School of Electrical and Computer Engineering of the University of Campinas in partial fulfillment of the requirements for the degree of Doctor in Electrical Engineering, in the Area of Computer Engineering.

Tese apresentada à Faculdade de Engenharia Elétrica e de Computação da Universidade Estadual de Campinas como parte dos requisitos para a obtenção do título de Doutor em Engenharia Elétrica, na Área de Engenharia da Computação.

Supervisor/Orientador: Prof. Dr. Mario Jino

Este exemplar corresponde à versão final da Tese defendida por Bruno Luis Pires de Azevedo e orientada pelo Prof. Dr. Mario Jino.

CAMPINAS
2019

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca da Área de Engenharia e Arquitetura
Elizangela Aparecida dos Santos Souza - CRB 8/8098

Az25p

Azevedo, Bruno Luis Pires de, 1977-
A proposal for traceability in software development / Bruno Luis Pires de Azevedo. – Campinas, SP : [s.n.], 2019.

Orientador: Mario Jino.
Tese (doutorado) – Universidade Estadual de Campinas, Faculdade de Engenharia Elétrica e de Computação.

1. Rastreabilidade. 2. Software (Engenharia). I. Jino, Mario, 1943-. II. Universidade Estadual de Campinas. Faculdade de Engenharia Elétrica e de Computação. III. Título.

Informações para Biblioteca Digital

Título em outro idioma: Uma proposta para rastreabilidade no desenvolvimento de software
Palavras-chave em inglês:
Traceability
Software (Engineering)
Área de concentração: Engenharia de Computação
Titulação: Doutor em Engenharia Elétrica
Banca examinadora:
Mario Jino [Orientador]
Nandamudi Lankalapalli Vijaykumar
Selma Shin Shimizu Melnikoff
Léo Pini Magalhães
Romis Ribeiro de Faissol Attux
Data de defesa: 14-11-2019
Programa de Pós-Graduação: Engenharia Elétrica

Identificação e informações acadêmicas do(a) aluno(a)
- ORCID do autor: https://orcid.org/0000-0003-3640-3250
- Currículo Lattes do autor: http://lattes.cnpq.br/4274571546762055

Universidade Estadual de Campinas
Faculdade de Engenharia Elétrica e de Computação

Bruno Luis Pires de Azevedo

A Proposal for Traceability in Software Development

Uma Proposta para Rastreabilidade no Desenvolvimento de Software

Banca Examinadora:

- Prof. Dr. Mario Jino
  Universidade Estadual de Campinas
- Prof. Dr. Nandamudi Lankalapalli Vijaykumar
  Instituto Nacional de Pesquisas Espaciais
- Prof.ª Dra. Selma Shin Shimizu Melnikoff
  Universidade de São Paulo
- Prof. Dr. Léo Pini Magalhães
  Universidade Estadual de Campinas
- Prof. Dr. Romis Ribeiro de Faissol Attux
  Universidade Estadual de Campinas

A ata da defesa, assinada pelos membros da Comissão Examinadora, consta no SIGA/Sistema de Fluxo de Dissertação/Tese e na Secretaria do Programa da Unidade.

Campinas, 14 de novembro de 2019

Acknowledgements

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Finance Code 001 – Programa de Excelência Acadêmica (PROEX) 0487 – Processo 1142618.

Resumo

Rastreabilidade tem sido um tópico de pesquisa no desenvolvimento de software por pelo menos 40 anos, sendo adicionada a muitos padrões, como o DOD-STD-2167A e o IEEE 830-1998. Este último, por exemplo, afirma que uma boa especificação de requisitos de software deve ser rastreável. A rastreabilidade fornece muitos benefícios para projetos de software, tais como: identificação das razões para decisões de design, prevenção de problemas de dependência, identificação de responsabilidades em um projeto, estimação de impacto e de custo de modificações, e medição do progresso de desenvolvimento. Sucintamente, a rastreabilidade permite a geração de um produto de melhor qualidade. Dois principais focos surgiram na literatura nos últimos anos: desenvolvimento baseado em modelo e geração automática de rastros. O primeiro trata da modelagem de rastreabilidade, definindo relações e elementos de um projeto; o segundo trata da descoberta automática de relações entre elementos. Vários conceitos foram definidos até agora, como rastreabilidade bidirecional, rastreabilidade de especificações pré e pós-requisitos, rastreabilidade horizontal e vertical, e rastreabilidade explícita e implícita. Embora haja um consenso geral sobre a maioria dos conceitos relacionados a rastreabilidade, há uma falta de consenso sobre como, e o quê, deve ser rastreado; não há consenso sobre: quais relações são relevantes para os projetos de desenvolvimento de software, quais elementos devem ser rastreados, como as mudanças nos elementos de um projeto afetam as relações existentes, ou como atualizar as relações dadas certas mudanças. Os modelos de rastreabilidade visam responder a essas questões fornecendo um padrão para ser usado como uma guia em projetos de desenvolvimento de software; entretanto, não há consenso sobre o que um modelo deve conter. Existe uma variedade de modelos, cada um considerando diferentes tipos de relações, elementos, e possuindo diferentes focos. Além disso, a maioria dos modelos possui problemas que tornam o seu uso difícil, ou até mesmo impossível; por exemplo, existem modelos que não descrevem – suficientemente ou em nada – as ligações de rastreabilidade que propõem. Este trabalho visa a ajudar nesta questão, fornecendo uma contribuição dupla: um modelo de referência para criar e avaliar modelos de rastreabilidade, e um metamodelo abrangente, construído em cima do modelo de referência, para adicionar rastreabilidade a projetos de desenvolvimento de software. Nosso Modelo de Referência para rastreabilidade define os elementos básicos em um modelo de rastreabilidade e define conjuntos básicos de: ações, tipos de ligações, tipos de artefatos, e processos. Propriedades necessárias para o conjunto de tipos de ligações e o conjunto de tipos de artefatos também são fornecidas. Nosso Metamodelo para rastreabilidade é composto por: um modelo conceitual descrevendo e organizando os elementos de rastreabilidade; um conjunto de tipos de artefatos representando as atividades definidas no Modelo de Referência, além de um conjunto de tipos de artefatos criados para registrar decisões de design; um conjunto de tipos de ligações que modelam diferentes relações de rastreabilidade; e um conjunto de processos para garantir a consistência de projetos.

Abstract

Traceability has been a topic of research in software development for at least 40 years, being added to many standards, such as the DOD-STD-2167A and the IEEE 830-1998. The latter, for instance, states that a good software requirements specification should be traceable. Traceability provides many benefits to software projects, such as: identification of the reasons for design decisions, avoidance of dependency issues, identification of accountability in a project, estimation of impact and cost of modifications, and measurement of development progress. Succintly, traceability allows the generation of a better quality product. Two main focuses have emerged in the literature in recent years: model-based development and automated trace generation. The former concerns modeling traceability by defining relations and elements in a project; the latter concerns automatic discovery of relations between elements in a project. Several concepts have been defined so far, such as bidirectional traceability, pre and post-requirements specification traceability, horizontal and vertical traceability, and explicit and implicit traceability. While there is general consensus on most concepts related to traceability, there is a lack of consensus on how, and what, should be traced; there is no consensus on which relations are relevant for software development projects, which elements should be traced, how changes in elements of a project affects existing relations, or how to update relations given certain changes. Traceability models aim to answer these questions by providing a standard to be used as a guide in software development projects; however, there is no consensus on what a model should contain. There is a variety of models, each considering different types of relations, elements, and having distinct focuses. Also, the majority of models have issues which makes them difficult or even impossible to use; for instance, there are models which do not describe – sufficiently or at all – traceability links which they propose. This work aims to help in this issue by providing a twofold contribution: a reference model for creating and evaluating traceability models, and a comprehensive metamodel, built on top of the reference model, to add traceability to software development projects. Our Reference Model for traceability defines the basic elements in a traceability model and defines basic sets of: actions, link types, artifact types, and processes. Necessary properties for the sets of link types and artifact types are also provided. Our Metamodel for traceability is composed of: a conceptual model describing and organizing the elements of traceability; a set of artifact types representing the activities of the Reference Model, plus a set of artifact types created to record design decisions; a set of link types modeling different traceability relations; and a set of processes to ensure project consistency.

Contents

I Introduction and Literature Review **12**

1 Introduction **13**

1.1 Benefits of Traceability . . . 15
1.2 Terminology . . . 16
1.3 Contribution . . . 17
1.4 Structure . . . 18

2 Literature Review **20**

2.1 Research Questions . . . 20
2.2 Protocol . . . 20
2.2.1 Data Sources and Research Strategy . . . 21
2.2.2 Inclusion and Exclusion Criteria – Initial Selection . . . 21
2.2.3 Paper Analysis Strategy . . . 22
2.3 Search Results . . . 22
2.3.1 Results of the Initial Selection . . . 23
2.3.2 Most Relevant Papers – Final Selection . . . 23
2.4 Conclusions . . . 28
2.4.1 Issues Identified in the Literature . . . 28
2.4.2 Evaluating the Final Selection of Papers . . . 29

II A Reference Model for Traceability **32**

3 Reference Model **33**

3.1 Basic Elements in a Traceability Model . . . 34
3.2 Basic Actions on Artifacts . . . 35
3.2.1 Defining Each Basic Action . . . 36
3.2.2 Why a Reference Model Should Establish the Basic Actions on Artifacts . . . 37
3.3 Link Types . . . 38
3.3.1 Link Types Must Be Described . . . 38
3.3.2 Basic Link Types . . . 38
3.3.3 About the Basic Link Types . . . 43
3.3.4 Justifying the Relations Modeled by the Reference Model . . . 43
3.4 Artifact Types . . . 45
3.4.1 Pre-Requirements . . . 46
3.4.2 Requirements . . . 46
3.4.3 Design . . . 46
3.4.4 Implementation . . . 46

3.4.5 Verification & Validation and Testing . . . 46
3.4.6 Actors Are Not Artifacts But... . . . 46
3.5 Ensuring Traceability and System Consistency . . . 47
3.5.1 Basic Processes . . . 47
3.6 Necessary Properties of Link Types and Artifact Types . . . 48
3.6.1 Comprehensiveness . . . 48
3.6.2 Specificity . . . 48
3.6.3 Artifact Coverage . . . 49
3.6.4 About the Necessary Properties . . . 49

4 Using the Reference Model 50
4.1 Evaluation Strategy . . . 50
4.2 Ramesh and Jarke . . . 51
4.2.1 Classification of the Link Types . . . 52
4.2.2 Description Level of the Link Types . . . 52
4.2.3 Evaluation of the Necessary Properties of Link Types . . . 53
4.2.4 Classification of the Artifact Types . . . 53
4.2.5 Description Level of the Artifact Types . . . 55
4.2.6 Evaluation of the Necessary Properties of Artifact Types . . . 55
4.2.7 Processes . . . 55
4.2.8 Strengths & Limitations . . . 55
4.3 Goknil et al. . . . 56
4.3.1 Classification of the Link Types . . . 57
4.3.2 Description Level of the Link Types . . . 57
4.3.3 Evaluation of the Necessary Properties of Link Types . . . 57
4.3.4 Classification of the Artifact Types . . . 58
4.3.5 Description Level of the Artifact Types . . . 58
4.3.6 Evaluation of the Necessary Properties of Artifact Types . . . 58
4.3.7 Processes . . . 58
4.3.8 Strengths & Limitations . . . 59
4.4 Closing Remarks . . . 59

III A Metamodel for Traceability 60

5 An Overview of the Metamodel 61
5.1 Traceability Space . . . 61
5.2 Artifact Types . . . 62
5.3 Link Types . . . 62
5.4 Processes . . . 62
5.5 Outline by Chapter . . . 62

6 The Traceability Space 63
6.1 Interactions Between the Elements of Each Space . . . 65
6.2 Actors Space . . . 65
6.2.1 Meta-Actor . . . 65
6.3 Rules Space . . . 66
6.4 Processes space . . . 66
6.5 System Space . . . 67

6.5.1 Not Using the Homologation Process 68
6.6 Traceability Space: Definition 68
6.7 Actions: Definitions 70
6.7.1 Modification 71
6.7.2 Removal 71
6.7.3 Application 71
6.7.4 Decomposition 72
6.7.5 Reification 72
6.7.6 Creation 72
6.7.7 Homologation 73
6.7.8 Activation 73
6.8 The Traceability Space and Metrics 74

7 Artifact Types 75
7.1 Non-Rationale Artifact Types 75
7.2 Rationale Artifact Types 76
7.2.1 Rationale for Modification 76
7.2.2 Rationale for Removal 76
7.2.3 Rationale for Decomposition 76
7.2.4 Rationale for Homologation or Rejection 77
7.2.5 Rationale for Creation 77
7.2.6 Rationale for Application 77
7.2.7 Rationale for Activation 78

8 Relations Modeled as Link Types 79
8.1 Relations Modeled as One or Two Link Types? 80
8.2 Link Types 81
8.2.1 Evolution Link Types 81
8.2.2 Constraint Link Types 83
8.2.3 Accountability Link Types 86
8.2.4 Permission Link Types 91
8.2.5 Characterize Action Link Types 94
8.2.6 Action Outcome Link Types 101
8.2.7 Composition Link Types 102
8.3 Closing Remarks 104

9 Processes for Traceability 105
9.1 Homologation Process 106
9.1.1 Algorithm 106
9.1.2 Textual Description 108
9.1.3 Assessing the Impact of Homologating a Rationale for Modification or a Rationale for Removal 111
9.1.4 Why is the Homologation Process Useful? 117
9.2 Modification Process 117
9.2.1 Algorithm 117
9.2.2 Textual Description 118
9.3 Decomposition Process 119
9.3.1 Algorithm 119
9.3.2 Textual Description 121

9.4 Creation Process . . . 123
9.4.1 Algorithm . . . 123
9.4.2 Textual Description . . . 125
9.5 Removal Process . . . 126
9.5.1 Algorithm . . . 126
9.5.2 Textual Description . . . 127
9.6 Activation Process . . . 127
9.6.1 Algorithm . . . 128
9.6.2 Textual Description . . . 129
9.7 Application Process . . . 130
9.7.1 Algorithm . . . 130
9.7.2 Textual Description . . . 131
9.8 Processes and Permissions . . . 131
9.8.1 Creating and Updating Permissions . . . 131
9.9 Change Impact Analysis and Processes . . . 132

10 Using the Metamodel **133**
10.1 Conflicts Between Requirements . . . 133
10.1.1 Removing Requirement R18 . . . 135
10.1.2 Modifying Requirement R74 . . . 136
10.1.3 Removing Requirement R18 and Requirement R74 . . . 137
10.2 Closing Remarks . . . 138

IV Conclusion **139**

11 Conclusions, Limitations, and Future Work **140**
11.1 Reviewing the Contribution . . . 141
11.2 Simplifying the Metamodel . . . 143
11.3 Future Work and Limitations . . . 144

Bibliography **146**

A List of Papers of the Initial Selection **153**

Part I

Introduction and Literature Review

Chapter 1

Introduction

Traceability is the ability to keep track of elements of a system throughout its life cycle [26]; this involves the knowledge of the origins of each element and the rationale for its existence. Elements relate to other elements in many ways; to add traceability information is to expose such relations.

Traceability has been a topic of research in software development for at least 40 years. The concept first appears in a paper by Randell [55] published in a NATO conference in 1968; the author examines computer system design methodologies and praises three projects for making the systems being designed “contain explicit traces of the design process”. Eight years later, the term *traceability* is first used in a survey of the state of art and future trends of software engineering by Boehm [7]: ”Other capabilities are currently missing, such as support for configuration control, traceability to design and code, detailed consistency checking, and automatic simulation generation”. Traceability is finally defined in 1978 by Greenspan and McGowan [27]: “This is the property of a system description technique which allows changes in one of the three system descriptions – requirements, specification, implementation – to be traced to the corresponding portions of the other descriptions. The correspondence should be maintained throughout the lifetime of the system”.

By the 1980s, traceability started being added to many standards [11], such as the United States defense standard DOD-STD-2167A, published in 1988. Research on the topic begins to grow in the 1990s; Cleland-Huang et al. [11] atribute this growth as being driven by two new events: the IEEE International Symposium on Requirements Engineering and the IEEE International Conference on Requirements Engineering. Several authors have contributed to a better understanding of traceability in this decade: Ramesh and Edwards [52] highlight several important issues in the subject; Gotel and Finkelstein [26] analyze the problem, identifying the need for pre-requirements traceability; Pfleeger and Bohner [50] propose traceability graphs and metrics for vertical and horizontal traceability in the context of impact analysis; Lindvall and Sandahl [40] seek to characterize practical implications of traceability by studying an industrial-scale project, and propose a two-dimensional classification of traceability; Ramesh et al. [51] propose a simple framework for developing traceability models containing a general traceability model and presents a detailed traceability model for requirements management; among others. Published in this same decade, the IEEE Standard 830-1998 [34] defines that a good software require-

ments specification (SRS) should be traceable, and recommends backwards and forwards traceability as desirable properties; a definition is also provided: "An SRS is traceable if the origin of each of its requirements is clear and if it facilitates the referencing of each requirement in future development or enhancement documentation".

Two main focuses can be found in the research of the last two decades on the subject: model-based development [24, 13, 25, 2] and automated trace generation [9, 56, 10, 12]. The former concerns modeling traceability, defining relations and elements of traceability in a project; the latter concerns automatic discovery of relations between elements in a project. Many contributions for the topic were published from the 00s to recent day, such as: Ramesh and Jarke [53] propose two traceability models based on extensive empirical work; Goknil et al. [21] provide a traceability model with formally defined traceability links; Mäder et al. [41] highlight a few basic concepts which traceability models should take into account; Berenbach and Wolf propose a traceability model integrating different modeling techniques; Cleland-Huang et al. [11] composed a book containing various works discussing concepts and the state-of-art in traceability; Dubois et al. [16] propose a model for requirement traceability connecting the requirement model, the solution model, and the V&V model; Rochimah et al. [57] assess several traceability approaches concerning their contributions to software evolution; Mäder et al. [43] propose a link model for the Unified Process and a set of verification rules for links; Galvao and Goknil [19], and Winkler and Pilgrim [70] present surveys on the topic of model-driven traceability; among many others.

Several concepts have been defined so far: bidirectional traceability refers to the ability to trace forwards and backwards between elements of different development activities [52]. Pre-requirements specification traceability and post-requirements specification traceability refers to tracing related pre-requirements and post-requirements elements, respectively [26]; i.e., the identification of the elements which contributed to the creation of the requirements, and the identification of the elements created from reified information from the requirements. Traceability matrix refers to a matrix recording all the traceability links between elements in a project [11]. Horizontal and vertical traceability refers to the relations between elements of the same development activity, or between distinct development activities, respectively [52]; for instance, a relationship between two requirements is considered horizontal traceability and a relationship between a requirement and a design element is considered vertical traceability. Manual tracing, automated tracing, and semi-automated tracing refers to traceability being established by a human tracer, by an automated methods and tools, and by a combination of automated methods and tools, and humans, respectively [11]. However, there is no definition consensus for some concepts. For instance, explicit and implicit traceability links may describe: links which are identified by using models, and links which are generated automatically, respectively [48]; or links which are explicitly established, and the relations existing between the elements [42].

While there is general consensus on most concepts related to traceability, there is no consensus on how and what should be traced; for instance, consider the following questions: which relations are relevant for software development projects? Which elements should be traced? How changes in elements of a project affects existing relations? How to update relations given certain changes? These are some of the relevant questions which

should be answered when using traceability in a development project. Traceability models aim to answer these questions by providing a standard to be used as a guide in software development projects. However, there is no consensus on what a model should contain, or trace. There is a variety of models, each considering different types of relations, elements, and focuses. Also, the majority of models have issues which makes them difficult, or even impossible, to use; for instance, there are models which do not describe – sufficiently or at all – traceability links which they propose.

This work aims to help by providing a twofold contribution: a reference model for creating, and evaluating, traceability models, and a comprehensive metamodel, built on top of the reference model, to add traceability to software development projects.

1.1 Benefits of Traceability

A relevant question when evaluating if traceability should be added to a project is: what are the benefits of having traceability? Traceability provides many benefits, some of which will be discussed in this section.

Traceability enables the identification and collection of reasons for design decisions. Decisions are made during the development process; sometimes, there may be multiple options to choose from. It may not be feasible to choose all of them; they may be exclusive, they may contradict each other, or it may be too costly to implement all existing options. Given enough time, the rationale for these decisions may be lost. For instance, suppose a project where a requirement violates a newly added business rule. In order to resolve this conflict, it is decided to modify the requirement according to this new rule. At the time this decision is made, the group of people involved may be able to justify why they modified the requirement; they will explain how the previous version of the requirement violated a certain business rule. However, after a while, this information may be forgotten and, consequently, the modification will lose its justification. Traceability can solve this problem by tracing the rationale for design decisions throughout the product life cycle.

Avoidance of dependency issues is another advantage provided by traceability. Traceability enables tracing dependency relations between artifacts; by maintaining this information throughout the product life cycle, it is possible to avoid dependency issues due to changes in the project. For instance, consider one artifact which depends on another artifact. The knowledge of this relation may avoid the removal of the necessary artifact, or may result in its replacement by another artifact which will keep the project consistent.

Traceability enables accountability for actions in a project. For instance, suppose an artifact which will be modified. The group of people in charge of this task can discover, through traceability links, those responsible for the most recent modification, or those who created the artifact. This information may facilitate their work; exchanging information with those other actors may generate greater knowledge of the artifact and possibly reduce the insertion of defects in the system.

The identification of who performed an action also enables rewarding good actions or recognizing the need for reallocation or personnel training. In the former, relevant contributions may be identified and those accountable may be rewarded accordingly; in

the latter, contributions having serious flaws may indicate the need to reallocate personnel to better suited work or the need to provide further training.

Accountability for actions is also useful to identify overworked personnel and reallocate roles accordingly. For instance, having information that a group of developers is accountable for significantly more artifact creations than other developers enables reallocating personnel to reduce their workload.

Traceability facilitates the estimation of the impact of a change; it enables identification of the elements which will be affected by a change by tracing relations between them. To exemplify, consider a modification to be performed on a requirement; it is possible to find the design and code artifacts which originated from the requirement, and consequently which may be affected, by traversing the relevant traceability links. Hence, traceability is essential for change impact analysis.

Traceability also enables checking if all requirements have been implemented, and if they were implemented correctly. If each requirement can be traced to code artifacts, it is a matter of verifying the corresponding artifacts. In the same way, it is possible to traverse the links in the opposite direction, and find out the origins of each artifact. The former ensures that all requirements were implemented; this is specially useful for safety-critical products. The latter provides rationale for the existence of the artifact.

By tracing artifacts which originated from requirements, traceability also enables the measurement of progress during development; i.e., it makes possible to find out how far the information from a requirement has evolved in the development process. For instance, given a set of requirements, by traversing the relevant links to code artifacts, it is possible to discover how many of the requirements are: completely implemented, partially implemented, and not implemented yet. This information may be used to estimate the completion status of the project.

Traceability may be used to perform assessments on the project. For instance, an evaluation of elements by its connection degree may reveal highly complex elements. These may be analyzed to find out if there is too much concentrated functionality; if so, the identified elements may be broken into smaller parts.

In conclusion, there are many benefits to using traceability in a project. Traceability enables the generation of a higher quality product, and any added costs can be recovered due to the benefits it provides.

1.2 Terminology

In this section, we define a few terms which are used throughout this text; some of these terms are defined in more detail later on.

Definition 1.2.1. Traceability is the ability to track elements in a project, and their relations, throughout its life cycle. Elements relate to other elements in many ways; to add traceability is to expose such relations.

Definition 1.2.2. A traceable element, or element, is a unit of information; e.g., a use case diagram, a code fragment, a requirement document, a method in a class, etc. The granularity of the element defines

Definition 1.2.3. An artifact type is a conceptual representation of an element from a project; i.e., requirement as a concept is an artifact type. An existing requirement in a project is not an artifact type.

Definition 1.2.4. A traceability artifact, or artifact, is an instance of an artifact type. Each artifact in a project belongs to a specific artifact type; e.g., a requirement in a project which describes a desired feature is an artifact belonging to the requirement artifact type.

Definition 1.2.5. A relation describes how two artifacts are connected; how they affect, or are relevant, to each other.

Definition 1.2.6. A link type is a conceptual representation of a relation between two artifacts; e.g., a relation of dependency between artifacts may be modeled by a link type which represents this relation.

Definition 1.2.7. A traceability link, or link, is an instance of a link type. Each traceability link in a project belongs to a specific link type; e.g., consider two artifacts which have a relation of dependency between them; a link, belonging to a link type which models this relation, connects these two artifacts.

Definition 1.2.8. To trace an element is the ability of finding an element by traversing traceability links.

Definition 1.2.9. A traceability process, or process, is a mechanism to ensure consistency after changes; it is composed of an action and instructions to ensure consistency given this action.

Definition 1.2.10. Traceability information is information which relates to traceability; e.g., the semantics of a traceability link is traceability information.

1.3 Contribution

Our contribution is twofold: we propose a Reference Model for traceability and a Metamodel for traceability. Also, several new concepts are defined throughout the text.

A literature review was done to find out the past and current state of research in modeling traceability for software development; it revealed common issues found in traceability models. The Traceability Reference Model intends to help with these issues. The Traceability Metamodel implements the Reference Model and expands substantially on what it defines.

The Reference Model defines the basic elements in a traceability model and defines basic sets of: actions, link types, artifact types, and processes. Necessary properties for the sets of link types and artifact types are also provided. The Reference Model has two distinct uses: it may be used to evaluate traceability models or it may be used to create traceability models. The Reference Model is used to evaluate two relevant contributions in the literature and as a basis to create our Metamodel for traceability.

The Metamodel is composed of: (i) a conceptual model describing and organizing the elements of traceability – the conceptual model and the actions contemplated by the

Metamodel are formally defined; (ii) a set of artifact types representing the activities of the Reference Model, plus a set of artifact types created to record design decisions; (iii) a set of link types modeling different traceability relations – each link type is formally defined and semantically described to avoid ambiguities in practical use; and (iv) a set of processes to ensure project consistency – each process is described by an algorithm and by a textual description. The Metamodel has one main use: to add traceability to a software development project. We apply the Metamodel on requirements from a course management system as a proof of concept.

1.4 Structure

This dissertation is structured as follows.

Chapter 2 describes the literature review; it contains the following sections: Section 2.1 provides the goals of the review; Section 2.2 details the protocol used to structure the review; Section 2.3 shows the results obtained by using the protocol; and conclusions for the review are shown in Section 2.4.

Chapter 3 details the Reference Model; it contains the following sections: Section 3.1 summarizes the basic elements of a traceability model; Section 3.2 describes the set of basic actions; Section 3.3 describes the set of basic link types; Section 3.4 details the set of basic artifact types; Section 3.5 discusses how to ensure consistency in a project and lists the minimum set of processes a traceability model should have; and Section 3.6 establishes three necessary properties of sets of link types and sets of artifacts types for a traceability model.

Chapter 4 shows an application of the Reference Model – the Reference Model is used to evaluate two contributions found in the literature; it contains the following sections: Section 4.1 specifies the strategy used to perform the evaluation; Section 4.2 shows the evaluation of the contribution by Ramesh and Jarke; Section 4.3 shows the evaluation of the contribution by Goknil and other authors; and Section 4.4 provides a few closing remarks about the evaluations.

Chapters 5–9 describe the Metamodel. Chapter 5 outlines the main components of the Metamodel. Chapter 6 describes and defines the conceptual model; it contains the following sections: Section 6.1 details the interactions between the basic elements of the Metamodel, given the conceptual model; Sections 6.2–6.5 describe the elements of the conceptual model; Section 6.6 defines the conceptual model; Section 6.7 defines the basic actions used in the Metamodel; and Section 6.8 briefly discusses possible metrics enabled by the Metamodel. Chapter 7 details the artifact types of the Metamodel; it contains the following sections: Section 7.1 lists the non-rationale artifact types of the Metamodel, and Section 7.2 defines the rationale artifact types of the Metamodel. Chapter 8 describes the link types of the Metamodel; it contains the following sections: Section 8.1 discusses two possible ways to model relations between artifacts, Section 8.2 defines each link type of the Metamodel, and Section 8.3 provides closing remarks on the subject. Chapter 9 describes the processes of the Metamodel; it contains the following sections: Sections 9.1–9.7 describe each process of the Metamodel; Section 9.8 summarizes necessary updates to

links in a project, given certain processes; and Section 9.9 discusses how processes relate to change impact analysis.

Chapter 10 shows a partial application of the Metamodel on requirements of a course management system; it contains the following sections: Section 10.1 shows the issues found and how to fix them by using the Metamodel; and Section 10.2 provides a few closing remarks.

Chapter 11 provides our conclusions about the work done; it contains the following sections: Section 11.1 provides a review of our contribution divided by chapter; Section 11.2 provides some observations about simplifying the Metamodel; and Section 11.3 discuss current limitations of the contribution and possibilities for future work.

Chapter 2

Literature Review

A literature review was performed to find out the past and current state of research in modeling traceability for software development.

This chapter is structured as follows: Section 2.1 provides the goals of the review; Section 2.2 details the protocol used to structure the review; Section 2.3 shows the results obtained by using the protocol; and Section 2.4 shows our conclusions.

2.1 Research Questions

This literature review was done to answer the following questions:

1. What contributions are part of the history of modeling traceability for software development?

2. What is the state of the art in modeling traceability for software development?

3. What shortcomings and/or limitations are there in current traceability models?

The first two questions were intended to generate knowledge on traceability. The third question is aimed at unveiling shortcomings and/or limitations for which we could propose solutions, or create a discussion; this question is the focus of this chapter, since it led to our contribution.

2.2 Protocol

The protocol used in the literature review is composed of four parts: (i) definition of data sources, (ii) definition of the research strategy, (iii) definition of criteria for inclusion and exclusion of papers, and (iv) definition of the analysis strategy. The research strategy is used to search the data sources, and the papers are filtered according to the criteria using the analysis strategy.

2.2.1 Data Sources and Research Strategy

The following bibliographic databases were searched: ACM Digital Library [1], ScienceDirect [2], IEEE Xplore Digital Library [3], and Springer Link [4]. All journals and conferences of these databases were considered. Two searches were carried out: the first search considered all results until the end of 2015; two years later, these results were updated by a search considering all papers from January 2016 to December 2017. The results of both searches were then joined.

The terms used in the searches were updated iteratively several times. The knowledge gained by performing multiple search variations enabled the improvement of the search; for instance, we observed that searching for traceability models generated many results having information retrieval as main subject, since there are papers focusing on automatic generation of traceability from source code. This new knowledge enabled us to reduce the size of the initial selection by adding the exclusion of the related term.

The final set of terms is shown in the Table 2.1. The search terms used when updating the results of the ACM Digital Library and ScienceDirect had to be changed to reflect changes in their search engines; these are shown in Table 2.2. The search terms used for the IEEE Xplore Digital Library and Springer Link were not changed.

Table 2.1: Search Terms.

Database	Search Terms
ACM DL	(Abstract:traceability) and (Abstract:model or Abstract:metamodel or Abstract:framework or Abstract:modelling) and (not Abstract:retrieval), (Title:traceability) and (Title:model or Title:metamodel or Title:framework or Title:modelling) and (not Title:retrieval)
ScienceDirect	TITLE-ABSTR-KEY(traceability) AND (TITLE-ABSTR-KEY(model) OR TITLE-ABSTR-KEY(metamodel) OR TITLE-ABSTR-KEY(framework) OR TITLE-ABSTR-KEY(modelling)) AND NOT TITLE-ABSTR-KEY(retrieval)
IEEE XDL	(traceability AND (model OR metamodel OR framework OR modelling) AND NOT retrieval)
Springer Link	traceability AND (model OR metamodel OR framework OR modelling) AND NOT retrieval. Filters applied: Software Engineering and English.

2.2.2 Inclusion and Exclusion Criteria – Initial Selection

The main criteria for this initial selection were: papers containing traceability models, traceability frameworks, traceability links, traceability applied to software development

[1] `http://dl.acm.org/`.
[2] `http://www.sciencedirect.com/`
[3] `http://ieeexplore.ieee.org/`
[4] `http://link.springer.com/`.

Table 2.2: Updated Search Terms for the 2016-2017 Interval.

Database	Search Terms
ACM DL	recordTitle:(model metamodel framework modelling -retrieval +traceability), recordAbstract:(model metamodel framework modelling -retrieval +traceability)
ScienceDirect	TITLE-ABSTR-KEY(traceability) AND (TITLE-ABSTR-KEY(model) OR TITLE-ABSTR-KEY(metamodel) OR TITLE-ABSTR-KEY(framework) OR TITLE-ABSTR-KEY(modelling)) AND NOT TITLE-ABSTR-KEY(retrieval)[All Sources(Computer Science,Engineering)]

projects, properties of traceability link types, and traceability concepts. Papers not strongly focusing on traceability but having interesting link types were also selected.

Systematic reviews on the topic of traceability were also selected to identify relevant work in the literature and discarded afterwards; snowballing was not restricted to systematic reviews, being also done on other papers.

Abstracts, slides, very short versions of current papers were investigated to identify and evaluate their correspondent works and then discarded.

However, not every paper in this selection needed to have traceability as its main focus or to have traceability applied to software development; the goal of this initial selection was to find the most relevant work but also to learn more about the topic, even if the papers did not directly relate to our work. Therefore, as a secondary criteria, we considered: works loosely related to traceability but having interesting concepts; papers having traceability link types, even if they are not treated as such; traceability applied to other areas; papers focusing on design rationale, which is relevant for traceability; papers not focused on traceability but having novel relations between elements; papers containing requirements models; interesting actions on elements, e.g., decomposing requirements; metrics for traceability; traceability applied to domain-specific projects, e.g., safety.

In conclusion, relevant papers, plus papers which could add to our knowledge on the topic, and in related topics, were added to this initial selection.

2.2.3 Paper Analysis Strategy

The examination of papers adhered to the following sequence: evaluation of abstract, evaluation of the introduction and conclusion, evaluation of the complete article; if the evaluation was not enough to exclude the article, we proceeded to the next evaluation in the sequence. Most articles required evaluating the introduction and conclusion to be selected or excluded.

2.3 Search Results

Using the search terms and the data sources defined in the protocol, we obtained 1111, 497, 1092, and 3933 papers in ACM Digital Library, ScienceDirect, IEEE Xplore Digital

Library, and Springer Link, respectively, for a total of 6633 papers; from these, 6147 papers were found in the first search and 486 were found in the update. Table 2.3 shows the number of papers found organized by database and year interval.

Table 2.3: Search Results + Search Update Results.

Database	Results Until 2015	Results 2016-2017	Total
ACM DL	952	159	1111
ScienceDirect	449	48	497
IEEE XDL	920	172	1092
Springer Link	3826	107	3933

2.3.1 Results of the Initial Selection

Using the criteria and the analysis strategy described previously, the full set of 6633 papers was investigated; snowballing was also used to find relevant papers. We found a selection of 212 papers of direct interest; Table 2.4 shows the number of papers found organized by database. The complete list of papers in this selection can be found in Appendix A.

Table 2.4: Results of the Initial Selection.

Database	Initial Selection
ACM DL	21
ScienceDirect	27
IEEE XDL	94
Springer Link	70

This initial selection was used to find papers which are strongly related to our work but also to learn more about traceability; as mentioned previously, we also considered papers which added to our knowledge on traceability and related subjects.

The first criteria was used to find the most relevant papers, but many useful and interesting works were also found by using the secondary criteria; e.g., traceability and agile development [65, 6, 3], traceability in software product lines [14, 4, 18], a work identifying four types of dependency relations [71], traceability and safety critical systems [46, 49, 58], design rationale [63, 61, 64], traceability applied to detection of attacks by intrusion detection systems [33], traceability modeling social interactions [59], tools supporting traceability [29, 32, 31, 17], a seminal work [27], among many other contributions.

2.3.2 Most Relevant Papers – Final Selection

A deeper inspection was performed in the initial selection to find the papers most closely aligned to our work; papers containing traceability models and frameworks, basic concepts

of traceability, and distinctive traceability link types were selected. The following 22 papers were considered the most relevant to our research interest.

Concepts and important issues to be taken into account when developing a model for traceability are highlighted in [52] and [41].

Most papers in this selection contain models for traceability: a model derived from a real case scenario is presented in [54]; empirical approaches were used in [51] and [53] to create models; a case study was used to develop a model integrating traceability and Software Configuration Management in [45]; a model integrating different modeling techniques is described in [5]; in [13], a model relating the requirements model to the architecture model is proposed; a model and an approach for automatic generation of link types is described in [35]; a traceability management method for modeling traceability with Multi perspectives View is proposed in [20]; a feature-oriented traceability model for software product line development is presented in [60]; a four step traceability management process and a model for traceability are proposed in [28]; in [16], a model for requirement traceability linking the requirement model, the solution model, and the V&V model is described; a model which integrates textual specifications with UML specifications is proposed in [39]; a model-driven approach for supporting the evolution of design decisions is described in [44]; a link model for the Unified Process and a set of link verification rules are proposed in [43]; in [23], a metamodel having a set of formalized link types is proposed and is used in a subsequent work to support different goals: consistency checking [21], reasoning about requirements [22], validation of traces between requirements and architecture [24], and change impact analysis [25]; a framework containing mechanisms for model slicing and a model for traceability are proposed in [47].

2.3.2.1 Brief Description Plus Links and Artifact Types

Each paper listed previously is succintly described and, if the work has a traceability model, we list the link types and artifact types it proposes.

Ramesh and Edwards [52] highlights several important issues when developing a model for requirements traceability; these are: bidirectional traceability, criticality of requirements, design rationale, project tracking and management, accountability, humanware, documents/manuals, dependencies, horizontal and vertical traceability, and automated support for traceability.

Ramesh et al. [54] present a case study of a system development organization; the subject system has 75,000 code lines and 3,000 requirements. A traceability model derived from the case study is proposed. It contains artifact types and link types. Artifact types: Change Proposal, Rationale, Test, Simulation, Requirements, Stakeholder, Constraints, Design, Compliance Verification Procedure, System Components, Resources, External System, and Source Document. Link types: Initiates, Modify, Based on, Derive, Is a, Create, Satisfies, Verifies, Allocated to, Dictates, Defines, Assigned to, Interfaces/Depends on, Interfaces, and Reference.

Ramesh et al. [51] propose a framework for traceability models containing a high level traceability model, and presents a detailed traceability model for requirements management: the high level requirements traceability model and the detailed traceability model.

The detailed traceability model contains artifact types and link types. Artifact types: Assumptions, Rationale, Alternatives, Derived Requirement, Decisions, Issues, Critical Success Factors, Requirements, and Standards/Policies/Methods. Link types: based-on, supported-by, evaluates, leads-to, considers, is-a, tracking-by, generates, and validate.

Ramesh and Jarke [53] conducted empirical studies on 26 software development organizations and developed two reference models for traceability: a low-end model and a high-end model, for users of different profiles. Two case studies were performed using the proposed models. The low-end model contains four artifacts types and seven link types. The high-end model is composed of four submodels: Requirements Management submodel, Rational Submodel, Design Allocation Submodel, and Compliance Verification Submodel. The Requirements Management submodel contains fourteen artifact types and fourteen link types. The Rational Submodel contains eleven artifact types and seventeen link types. The Design Allocation Submodel contains eleven artifact types and thirteen link types. The Compliance Verification Submodel contains thirteen artifact types and six link types. The traceability link types used in the reference models are classified into four categories: Satisfaction links, Evolution links, Rational links, and Dependency links.

Berenbach and Wolf [5] propose an unified requirements model integrating different modeling techniques; it supports traceability in features modeling, use cases modeling, requirements analysis, requirements and hazard analysis, detailed requirements, and system design. The four models shown have nineteen artifact types and twenty five link types. The artifact types are: AbstractFeature, Feature, Stakeholder Request, Alternative, Use case, Stakeholder, Actor, Sequence Diagram, State Diagram, Activity Diagram, Class, Object, Requirement, Nonfunctional Requirement, Functional Requirement, Mitigation, Hazard, Cause, and SystemModel Element. The link types are: Requires, Conflicts, May become, Parents, Subfeatures, Alternative, Participating, Iniciating, Described by, Includes, Extends, Inherits, Expose in, Participates, Participating Objects, Detailed in, Constrained by, Constraints, Refines, Instance, Involved Entity, Target, Triggers, Mitigates, and Hazardous Element.

Dermeval et al. [13] propose an unified metamodel for architectural design decisions; it relates the requirements model to the architecture model allowing the estimation of the impact of requirements changes on the architecture. The metamodel contains eighteen artifact types and fifteen link types. The artifact types are: NFR, Contribution, Rationale, Alternative, Requirement, Functional, Stakeholder, DecisionDependency, Decision, DesignFragment, ArchitecturalArtifact, Artifact, ManagementArtifact, ImplementationArtifact, RequirementsArtifact, Consequence, OrganizationActo, and SystemActor. The link types are: contributedBy, contributionTarget, contributions, contributionSource, alternatives, rationales, proposes, isProposedBy, appends, dependencySource, dependencyTarget, consequences, produces, isRelatedTo, and modified.

Jirapanthong and Zisman [35] propose a rule-based approach allowing automatic generation of traceability link types between feature-based object-oriented documents (artifacts), intended to support product line engineering. A reference model for traceability having different types of relationships and documents, and an approach for automatic generation of traceability link types are described. The reference model contains eight artifact types and ten link types. The artifact types are: Feature Model, Subsystem Model,

Process Model, Module Model, Use Case, Class Diagram, Statechart Diagram, and Sequence Diagram. The link types are: Refines, Satisfies, Depends_on, Similar, Overlaps, Different, Contains, Evolves, Implements, and Encompasses. A prototype tool was also developed.

Mäder et al. [43] propose a traceability link model for the Unified Process and a set of traceability link verification rules; these rules allow semi-automatic establishment and verification of links for the Unified Process development projects. The model contains four link types: Refine, Realize, Verify, and Define.

El ghazi [20] proposes a traceability management method for modeling traceability with Multi perspectives View; it is composed of a metamodel used to describe traceability information and a process for guidance when constructing the model. The metamodel contains the following perspectives: Actors, Products, Process, Evolution, and Traceability Link; the latter perspective contains the following link types: Satisfaction, Dependency, Evolution, Rationale, Containment, and Contribution.

Shen et al. [60] propose a feature-oriented traceability model for software product line development; it provides traceability representations during the goal model, feature model, feature implementation model, and program implementations. The model contains the following link types, classified into two categories: intra-level and inter-level. The intra-level links are: Decompose, HasElement, subClassOf, and configDepend. The inter-level links are: Support, Dynamic-operationalize, Static-operationalize, Realize, and Instantiate.

Haidrar et al. [28] propose a framework for requirement traceability composed of a management process and a metamodel. The process is composed of four steps: specification process, traceability identification, links elicitation, and links generation. The metamodel contains the following link types: Development Link, Contribution Link, and Justification Link. The Development Link is specialized into four subtypes: Refine, DeriveReqt, Verify, and SatisfyReqt. The Contribution Link is specialized into two subtypes: ResOf and Modified. The Justification Link is specialized into one subtype: JustifiedBy.

Dubois et al. [16] propose a metamodel for requirement traceability linking three models: the requirement model, the solution model, and the V&V model. The metamodel contains the following link types: Copy, Derive, Refine and Decompose for the requirement model; Satisfy for the solution model; Verify for the V&V model. A case study was performed using an ABS system for a vehicle.

Nejati et al. [47] propose a framework for specifying and extracting design aspects relevant to safety requirements; to achieve this goal, two components are used: a metodology to determine traceability between design and safety requirements, and an algorithm to extract – given a safety requirement – the corresponding design fragment. The framework contains a metamodel for traceability containing twenty five artifact types and six link types. The artifact types are: Assumption, System Context, OCL Constraint, Environment Block, System Block, Parametrics, Standard, Recommended Practice, Stakeholder, Law/Regulation, Source, System-Level Safety Requirement, Block-Level Safety Requirement, Block-Level Requirement, Block-Level Safety-Relevant Requirement, Use Case, Mapping, Block, Block Relationship, Activity Partition, Block State, ACtivity

Node, Block Operation, and Activity Edge. The link types are: derive, derive/justify, refine, decompose, trace, and allocate.

Mäder et al. [41] highlight some concepts in which traceability models should take into account. The authors propose a simple way to define a traceability model and a set of analysis to be done in projects having traceability; these are: validating traces (ensuring traces represent correct relations), impact analysis and change propagation (changes being propagated only to dependent elements), coverage analysis (knowing cardinalities for links and checking if elements are missing), and relation count analysis (evaluating if elements are too connected to others, similar to cohesion in software development).

Mohan et al. [45] propose a traceability reference model aiming to integrate traceability and Software Configuration Management (SCM), focusing on change management. A tool was developed to help the integration of traceability and SCM. The reference model contains twenty four artifact types and sixteen link types. The artifact types are: Object, Stakeholder, Source, Process Object, Rationale, Activities, Product Object, Emails, Tech Notes, Document, Requirement Document, Code Document, Design Document, Requirement, Change Request, Component, Decision, Issue, Assumption, Argument, Alternative, Change, Version, and Configuration. The link types are: traces to, has role in, manages, documents, depends on, maps to, justifies, is tracked by, leads to, resolves, addresses, evaluates, selects, supports, opposes, and is derived from.

Letelier et al. [39] propose a requirements traceability metamodel which integrates textual specifications with UML specifications. The metamodel contains the following link types: Rationale Of, Trace To, Part Of, Responsible Of, Modifies, Validated By, Verified By, and Assigned To. The metamodel was defined as an UML profile allowing the application in a CASE tool; a configuration process – based on the proposed UML profile – for requirements traceability is provided. A tool was developed to implement the UML profile.

Malavolta et al. [44] propose a model-driven approach for supporting the evolution of design decisions. The approach includes: a metamodel for evolving design decisions; support for impact analysis for the evolution of artifacts through bidirectional traceability links between design decisions, requirements and artifacts; a technique for identification of design decisions. The metamodel contains the following link types: conflictsWith, alternatives, excludes, overrides, enables, constraints, relatedTo, dependsOn, subsumes, boundsTo, comprises, rationale, addresses, decision, raises, pertainsTo, responsibleFor, and interestedIn. An Eclipse plugin was developed to support the approach.

Goknil et al. [23] propose a metamodel for requirements models and an approach for customizing this metamodel to support different requirements specification techniques. The semantics of the concepts and of the relations are defined, allowing the detection of implicit relations and inconsistencies. The metamodel contains the following link types: Requires, Refines, Contains, and Conflicts. The link types are formalized using first-order logic. A case study was performed using an industrial mobile service application.

Goknil et al. [21] uses the previous work [23], adding a new link type, for consistency checking and to infer new relationships. A tool was developed to perform these activities [68]. The metamodel contains the following link types: Requires, Refines, PartiallyRefines, Contains, and Conflicts.

Goknil et al. [22] propose a “metamodeling approach which allows reasoning about requirements and their relations on the whole/composed models expressed in different modeling notations”. The proposed metamodel is specialized for Product-line and SysML, providing a common semantic domain, allowing reasoning on the composition of models expressed in these languages. The metamodel contains the link types defined in the previous work [21] plus the new link type Equals.

Goknil et al. [24] propose an approach for automatic trace generation and validation of traces between requirements and architecture. A metamodel is proposed, having the same link types as defined in a previous work [21] plus two new link types: AllocatedTo and Satisfies. The semantics of the proposed link types are used for generating and validating traces, supported by a tool based on the Eclipse Modeling Framework, the ATL model transformation language and the Maude tool set.

Goknil et al. [25] propose a change impact analysis approach for requirements. A classification of changes in requirements is provided. The formalized semantics of relationships and change types enables identification of alternative changes, identification of incorrect positive impacts, and checking of change consistency. A tool (TRIC) [68] was extended to support the proposed activities.

Significant contributions for traceability are given by these papers; there is a richness of traceability link types and artifact types. On the other hand, there are several shortcomings which are discussed in the next section.

2.4 Conclusions

We conclude our literature review by discussing issues identified in the literature and provide a table with an evaluation and comparison of the final selection of papers given these issues.

2.4.1 Issues Identified in the Literature

The following issues were identified: lack of description of link types and artifact types; lack of mechanisms to ensure consistency; missing link types for modeling common relations; missing common activities of development processes; and lack of concepts or properties of traceability.

2.4.1.1 Lack of Description

Most papers propose traceability links which are not described properly, leaving the reader to guess the semantics of each type just by its name or by a brief sentence. The practical application of a model is dependent on identifying the modeled elements in real life projects so that mapping is possible. If a model has link types which are not semantically described it may be difficult, or even not possible, to use the model. Describing link types reduces, or may even eliminate, ambiguity when assigning link types from a model to relations between elements in a project. For instance, consider a traceability model having two link types called “address” and “resolve”, shown in a graphical manner,

without a textual description. These link types seem to have a similar meaning, since the first one addresses an issue and the second one resolves an issue. Without a clear description it would be difficult, if even possible, to map these link types to relations in a project. Many models found in the literature have link types shown only graphically (e.g., [54, 13, 28, 39, 44, 45]).

The same issue of lacking semantics also happens with artifact types; most papers found do not describe the modeled artifacts.

2.4.1.2 Lack of Mechanisms to Ensure Consistency

Another issue identified in traceability models is the lack of mechanisms to ensure traceability consistency whenever a change happens; for instance, if a necessary artifact is removed, what steps should be taken to ensure the project is: (i) still working as it should, and (ii) not missing, or having incorrect, traceability links connecting its artifacts? The non-existence of such mechanisms make it difficult to use a model in a project. If a change happens, a process should be provided to ensure that traceability links are updated accordingly. A more comprehensive process would also ensure system consistence, since one of the benefits of using traceability is the avoidance of consistency issues; i.e., it would use traceability links to identify possible issues whenever changes are made in a project.

2.4.1.3 Missing Link Types

Most traceability models lack link types for modeling common relations, such as relations to represent accountability for actions and authorization for actions; hence, these models are not able to trace, for instance, who modified an artifact and if an actor is allowed to modify a specific artifact.

2.4.1.4 Missing Common Activities of Development

Most papers focus on the requirements engineering activity of development; they do not consider elements from other activities. For instance, verification, validation and testing is an activity rarely considered by traceability-themed papers; thus, relations and elements of this activity are usually not modeled.

2.4.1.5 Lack of Concepts

There is also a lack of concepts and properties for traceability in the studied literature; most papers consider the use of traceability but do not provide a conceptual basis for it, and for its elements.

2.4.2 Evaluating the Final Selection of Papers

In Table 2.5 we evaluate the papers highlighted in Section 2.3.2. Each paper is evaluated regarding five issues discussed previously: Link Description, Artifact Description, Diversity of Relations, Mechanisms to Ensure Consistency, and Development Activities.

Link description is classified as None (name only), Minimal (usually a sentence containing its name), Informal (informal description), or Formal (well-defined traceability relation, described formally); Artifact Description is analogous to Link Description; Diversity of Relations is classified as Lacking (e.g., do not consider common relations such as dependency) or Sufficient; Mechanisms to Ensure Consistency is classified as having (then listing which actions are considered) or not having; Development Activities lists which development activities are covered by the model.

Papers which are thoroughly evaluated in Chapter 3 are not discussed here (Ramesh and Jarke [53] and Goknil et al. [23, 21, 22, 24, 25]).

Table 2.5: Evaluation of the Final Selection.

Authors and Citation	Link Description	Artifact Description	Mechanisms to Ensure Consistency	Diversity of Relations	Development Activities
Ramesh et al. [54]	None	None	None	Lacking	Requirements and Implementation[1]
Ramesh et al. [51]	Minimal	Minimal	None	Lacking	Requirements
Berenbach et al. [5]	Minimal	Informal	None	Lacking	Requirements and Design
Dermeval et al. [13]	None	Minimal	None	Lacking	Requirements
Jirapanthong et al. [35]	Informal	None	None	Lacking	Design[2]
Mäder et al. [43]	Minimal	None	None	Lacking	Requirements, Design, and Implementation
El ghazi [20]	Informal	Minimal	None	Lacking	Requirements
Shen et al. [60]	Minimal	Minimal	None	Lacking	Requirements and Implementation[3]
Haidrar et al. [28]	None	None	None	Lacking	Requirements
Dubois et al. [16]	Informal	Minimal	None	Lacking	Requirements and V&V
Letelier et al. [39]	None	None	None	Lacking	Requirements, V&V, and Design
Malavolta et al. [44]	None[4]	Informal	None	Lacking	Design
Nejati et al. [47]	Informal[5]	None	None	Lacking	Requirements and Design
Mohan et al. [45]	None[6]	Minimal	None	Lacking	Requirements

[1]Contains an artifact named "design" modeling "design rationale" instead of artifacts of the design phase; has a "System Components" artifact, a very general artifact representing "hardware, software, humanware, manuals, policies, and procedures", to model implementation artifacts. [2]Plus Documents of the Feature-Oriented Reuse Method given it is a Feature-Oriented work. [3]Plus Feature Level and Feature Implementation Level; it is a Feature-Oriented work, thus it has different activities than classical software development. [4]Refers to another paper. [5]A very succint description, but provides the semantics of each link type. [6]Uses Ramesh et al. [53] model.

Ten out of fourteen papers have None or Minimal description of links; for instance, Mäder et al. [43] provides a very brief description of four link types, using the link name as part of the description, and Haidrar et al. [28] provides a succint description of two general link types while providing no description of the specific six link types.

Lack of description is even worse for artifact types; only two papers provide informal descriptions of the modeled artifact types: Berenbach et al. [5] and Malavolta et al. [44].

Every paper has Diversity of Relations classified as Lacking since they all lack at least one basic relation between elements, such as conflict or accountability relations. Dermeval et al. [13] and Berenbach et al. [5] are the only papers which take into account Stakeholders in their model; the latter does not have direct relations between Actors and Artifacts and the former and has a relation – represented by two link types in opposite directions – modeling who proposed a requirement. On the other side, it does not record accountability for modification or decomposition of requirements, for instance.

No papers in this selection provide mechanisms to ensure traceability consistency whenever a change happens.

Only two papers take into account the V&V phase of development: Dubois et al. [16] and Letelier et al. [39]; the latter considers verification of requirements, having the "verify" relation, and the former has "Validated by" and "Verified by" link types connecting to a "Test Specification" artifact.

Part II

A Reference Model for Traceability

Chapter 3

Reference Model

As shown in Chapter 2, most papers having traceability models: (i) lack well-defined traceability link types, i.e., a traceability link type is shown, or named, but never described semantically, making difficult or impossible to use; (ii) provide incomplete coverage of situations, i.e., link types do not model the most common relations such as dependency, accountability, or authorization; (iii) do not provide mechanisms to ensure consistency of traceability, i.e., processes are not provided to keep traceability and system consistency after system changes; (iv) consider only requirements traceability, ignoring other activities of the development process such as design or implementation, i.e., artifact types do not model the most common artifacts and the model ends lacking the corresponding relations; and (v) lack concepts and properties for traceability; i.e., most papers do not provide a conceptual basis for traceability.

To address issues such as these, a Reference Model is needed. A Reference Model for traceability helps by describing the basic elements and properties necessary to enable the practical use of a traceability model in a project; it may be used as a basis to create traceability models or to evaluate traceability models.

In this chapter, we propose a Reference Model for traceability. Our Reference model defines the basic elements in a traceability model and provides a conceptual model showing how these elements interact with each other. Given these basic elements, it defines basic sets for actions, link types, artifact types, and processes. These sets contain the minimum necessary elements for general use. Lastly, necessary properties for the sets of link types and artifact types are defined.

The basic elements are the building blocks of a traceability model. The conceptual model helps to visualize the roles of each element. The set of actions define the minimum set of actions which should be considered by a model; these are intrinsically related to the sets of link types, artifact types, and processes, and are used in their construction. The set of link types defines the minimum set of relations which should be considered by a model. The set of artifact types defines a set of traceable elements for general use; e.g., a design diagram or a requirement may be traceable elements in common development projects. The set of processes defines the minimum set of mechanisms which should be provided to ensure consistency in a project. Three basic properties are proposed for link types and artifact types; these are useful if a model creator wants to create these sets from scratch, or to evaluate sets of a traceability model.

This chapter is structured as follows: Section 3.1 summarizes the basic elements of a traceability model; Section 3.2 describes the set of basic actions; Section 3.3 describes the set of basic link types; Section 3.4 details the set of basic artifact types; Section 3.5 discusses how to ensure consistency in a project and lists the minimum set of processes a traceability model should have; and Section 3.6 establishes three necessary properties which sets of link types and sets of artifacts types should have in a traceability model.

3.1 Basic Elements in a Traceability Model

The three basic elements in a traceability model are artifacts, links, and processes. Actors interact with these elements directly or indirectly.

Definition 3.1.1. Artifacts are the workproducts of the software development process [66]. Artifacts can be classified into different types, each one modeling specific points of view of a project. Design documents, requirement documents and source code are examples of artifact types. An artifact can be a set of elements, a single element or part of an element; it also can be of any complexity or size as desired. For instance, an artifact can represent a source code document, a class or a single method of a class.

Definition 3.1.2. Links represent relationships between artifacts and artifacts, artifacts and actors, and actors and actors. Links can be classified into different types, each having specific semantics regarding the relationship between the elements it connects. For instance, a link connecting an actor and an artifact could express an author-product relationship.

Definition 3.1.3. Processes aim to maintain consistency of traceability information in the software system. Whenever an action happens, a process provides the instructions needed to maintain consistency. System consistency is directly related to traceability consistency; hence, a traceability process should also provide for system consistency. For instance, if a given artifact is removed and other artifacts depend on it, a process should provide the actions necessary to keep traceability and system consistency; i.e., the traceability elements should be updated accordingly.

Definition 3.1.4. Actors are representations of agents interacting with the project. Usually they are people but not necessarily, since other systems may interact with the target system.

Figure 3.1 shows how these elements relate to each other.

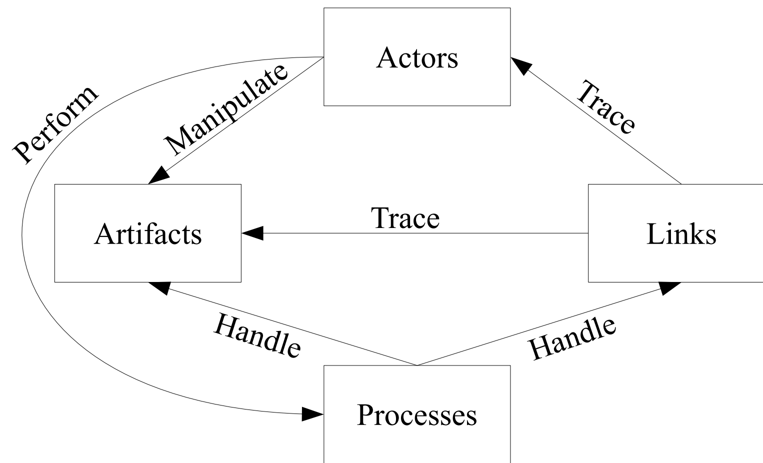

Figure 3.1: Conceptual model.

Actors perform processes; actors are agents who are able to execute processes and follow its instructions. Actors manipulate artifacts[1], performing actions on them. Processes handle artifacts and links; processes define the creation of new artifacts, generation of new links, deletion of links, etc. Links trace artifacts and actors; traceability links are the main element of traceability models, enabling the tracing of elements in a project.

3.2 Basic Actions on Artifacts

Actors cause changes in a project by manipulating artifacts. This manipulation can take various forms; for instance, to add a new feature to a project, an actor can create a new artifact or modify an artifact. Creation and modification are common actions which can happen in most projects. Removal is another common action, enabling actors to discard artifacts from the working project.

There are other actions which are also commonly needed. For instance, to test software it is necessary to apply test cases on source code. It may also be necessary to apply a checklist on the set of test cases to determine if everything is correct to proceed to the test execution phase. Hence, application of artifacts on other artifacts is another commonly needed action when developing software.

Artifacts sometimes are too big and complex, having a lot of unrelated information inside. For instance, a C++ class which goes through many modifications may end accomplishing several different functions, reducing its cohesion. To increase its cohesion it is necessary to break it into smaller classes. The action of decomposing bigger elements into smaller elements is another common action in software development.

Development processes work by gathering initial information and reifying this information continuously until it becomes working software. For instance, a requirement written in natural language becomes design artifacts written in a design language which, in turn, become code artifacts written in a programming language; i.e., the information located in the requirement was reified until becoming code. Each step in this sequence of evolutions

[1]From this point on, we will use the word manipulate to also mean “create”.

brings the information closer to the desired goal, the software. This action of reification is another typical action when developing software.

It may be desirable to check for quality and correctness of actions performed by actors on artifacts. The addition of artifacts to the working project without evaluating it may generate problems. This evaluation may be very simple, such as the actor who performed the action reviewing its own work or discussing the changes which were made with another actor. It also may be more thorough, by having an action verified by an independent group of actors before implementing it on the working project. To exemplify, consider a modification performed in a C++ class to add a new feature; this modification breaks another functionality which the performing actor is not aware of. The evaluation by another set of actors may identify this issue before its addition to the working project. We will denominate this action of evaluating changes before adding them to the working project as homologation. This is a very useful action in the context of traceability, since traceability has as one of its major benefits the avoidance of dependency and conflict issues by tracing relevant relations between artifacts. The homologation may also occur before an action takes place; i.e., the desired action is proposed and has to be homologated before it is done. This enables the assessment of the impact of the action, given the knowledge of the relations between elements provided by traceability, and consequently enables the informed decision on whether to proceed with it; it may be too costly or time consuming to perform the action, given the knowledge of the relations between the related artifacts. If it is decided to perform the action, the sequence of steps necessary to avoid consistency problems will be known during the homologation of this change. Homologating before an action takes place also reduces the work needed in a project, since actions can be discarded before being implemented. Succintly, homologation is an action which provides many benefits to projects when used together with traceability.

In conclusion, we establish the following seven actions as basic: creation, modification, removal, application, decomposition, reification and homologation; hence, a traceability model should take these into account when tracing actions on artifacts.

3.2.1 Defining Each Basic Action

Definition 3.2.1. Creation is the action through which an artifact comes into existence; an artifact is created by a group of actors.

Definition 3.2.2. Modification is the action of changing an artifact. To modify an artifact, a copy is created; the modification is done on this copy and the old version of the artifact is kept for future reuse or historical purposes.

Definition 3.2.3. Removal is the action of removing an artifact from the working project. Removal does not destroy an artifact; a removed artifact is kept for future reuse or historical purposes.

Definition 3.2.4. Application is the action of executing a procedure, defined in an artifact, on another artifact.

Definition 3.2.5. Decomposition is the action of decomposing an artifact into two, or more, artifacts. All the information from the original artifact is kept in the new artifacts,

no new information is added or removed. If adding or removing information is necessary, it can be done by concatenating the actions of modification and decomposition.

Definition 3.2.6. Reification is the action through which information evolves, becoming more structured and refined, in the development process, towards the goal of creating working software. A new artifact may be created by reifying information from other artifacts; thus, the reification action may be considered a special case of the creation action.

Definition 3.2.7. Homologation is the action of evaluating the addition of an artifact, or change, to the working project.

3.2.1.1 Are All Created Artifacts Reifications?

The actions of creation and reification are intrinsically connected since most artifacts created in a project are reifications of previously existing artifacts. For instance, a code artifact is a reification of parts of one, or more, design artifacts; a design artifact is a reification of parts of one, or more, requirements. However, not every new artifact in a project is a reification of another artifact; for instance, a test case is an artifact created to test artifacts in a project, and it is not a reification of existing information.

As mentioned previously, the reification action may be considered a particular case of the creation action, where a new artifact is constructed by reifying information of other artifacts which are not at the same reification level.

Definition 3.2.8. The reification level describes how close an artifact is to the final product; it is usually related to the development activity which produced the artifact. For instance, a code artifact is at a higher reification level than a requirement artifact.

3.2.1.2 Reifications Occur Vertically

A reification may only happen vertically; i.e., an artifact can not be reified into another artifact from the same reification level. For instance, a requirement can not be reified into another requirement. Modifying an artifact by adding more specific, detailed, information, is not reifying it; reification only happens when there is an evolution of the reification level.

3.2.2 Why a Reference Model Should Establish the Basic Actions on Artifacts

The establishment of the basic actions performed by actors on artifacts enables the definition of the basic link types and basic processes for traceability models. For instance, it is helpful to trace who performed a certain action; it allows an actor to contact the related actor to obtain more information about the action which was performed. Hence, the set of basic link types should have types concerning the authorship of this particular action. Without the establishment of a set of basic actions, it is not possible to define which actions should be traced when identifying authorship. The set of basic actions is

also needed when associating rationale for actions. The establishment of this set enables the definition of which link types are necessary to trace rationale.

The same is true for processes. Processes seek to maintain consistency in the project; to achieve this goal, each process is associated to a specific action being performed by actors on artifacts. For each particular action, the process provides the necessary instructions to avoid issues such as conflicts and unfulfilled dependencies. Therefore, the establishment of the common actions performed by actors enables the definition of the set of basic processes needed for a traceability model.

3.3 Link Types

A link type is a conceptual representation of a relationship between two elements. Traceability links are instances of link types. A project having traceability contains links connecting its elements, each link being of a link type.

3.3.1 Link Types Must Be Described

A traceability model must describe the semantics of each proposed link type to enable mapping each type to a relationship in the project. It may not be possible, or may be difficult, to use a traceability model on a project if link types are not described since the user will have to deduce the semantics of each type only by its name. Even if it is possible, ambiguity may generate errors when mapping link types to relationships.

This description does not have to be formal, but should be clear enough to reduce ambiguity and ease mapping each type to a relationship.

3.3.2 Basic Link Types

A software project may have many different types of relationships between its elements. Some link types may be relevant only to a very narrow set of domain-specific projects; other link types may be relevant to a larger set of projects. The latter is the set pertinent to this Reference Model.

A basic link type should be relatively common between projects and generic enough to enable the creation of more specific link types. Each link type in the Reference Model works as a general category from which a developer of a traceability model can use to create sets of specific link types within this category. The link types may also be used to evaluate a traceability model, identifying missing basic link types.

3.3.2.1 Evolution

A project is composed of information which changes and is propagated from one artifact to another. This information may become better structured and more detailed after being propagated, or it may just go through changes keeping the same structure and level of detail. As an example of the former, consider the information in a requirement which evolved until becoming a set of code artifacts. As an example of the latter, consider the creation of a new version of a code artifact after it is modified.

The evolution of information in artifacts, to more structured and refined versions, is typical in software development. As exemplified previously, requirements have information which evolves to become code; i.e., code is a reified version of information propagated from requirements. There may be intermediate artifacts, in which their information went through the same process. Each interconnected artifact in this sequence has this relation of propagation of reified information.

Another common relation happens between different versions of the same artifact; artifacts may go through many changes during its lifecycle generating a sequence of previous versions. Each connected artifact in this sequence has this relation of propagation of changed information, which is composed of smaller unchanged and changed blocks of information.

A traceability model lacking Evolution link types is not able to identify the origin and destination of information which was reified and propagated. It is not possible to identify from which requirements a certain code fragment evolved from or if a requirement was realized in other development phases. Consequently, it is difficult to do impact analysis, since we can not evaluate which artifacts are affected by a change; for instance, the addition of a new feature, changing a requirement, may impact several code artifacts, but without link types modeling these relations this information is lost. Another issue is losing previous versions of an artifact.

Link types of the Evolution link type convey that information was reified, or changed, and propagated from one artifact to another. It connects a previous version to a new version of an artifact which will have reified or changed information.

3.3.2.2 Constraint

Artifacts may constrain other artifacts in at least two ways: (i) an artifact may conflict with another artifact – e.g., a requirement which details the need for Apache servers and another requirement which details the need for IIS servers; (ii) an artifact may require another artifact – e.g, a requirement which details the need for IIS servers is dependent on another requirement detailing the need for the Windows Operating System in the machines for the servers. These are common relations found in software projects.

A traceability model lacking Constraint link types is not able to identify dependencies and conflicts between artifacts, risking the generation of consistency issues in projects. For instance, suppose a project which uses a traceability model without Constraint link types; an actor decides to remove a requirement which is necessary for other requirements, leaving the project inconsistent. This removal may not be identified until going to the next development phases, generating new costs and time spent fixing something which should be identified before the removal of the necessary requirement.

Link types of the Constraint link type enable the ability to identify artifacts which constrain other artifacts. It connects an artifact to another artifact which is constrained by it.

Types of Conflict: There are at least two types of conflict between artifacts: (i) contradictory information or (ii) redundant information. The former was exemplified previously;

the latter occurs when an artifact contains information sufficiently similar to that of another artifact. For instance, two requirements which detail the same functionality using different words; or, maybe one of these provides a more detailed account of the functionality and the other provides a less detailed account of the functionality. Nonetheless, the same information is provided; thus, there is redundant information in the project.

Usually, redundant information exists because of bad design or because of errors when creating or modifying artifacts. However, redundant information is not always an issue which must be fixed; there may be redundant information which is necessary or unavoidable. Each case must be evaluated individually to assess if it is an issue which needs to be dealt with.

3.3.2.3 Accountability

There is someone accountable whenever a change happens in a project; a single actor or a group of actors are responsible for changes performed during development. If a change happens and there is no record of who performed the change, useful traceability information is lost. Not having this information means not having: (i) identification of someone to be accountable for good or bad actions – good actions may be rewarded, and bad actions enables the recognition of a need for further training or maybe even identifying malicious actors inside the project; (ii) identification of someone who is able to explain details of the actions performed, enabling the transference of knowledge to a new group of actors – e.g., a code developer who previously performed a change may be invited to help with a new change which relates, or is similar, to the former change; (iii) a list of artifacts which an actor acted on – i.e., given an actor, being able to know which changes an actor took part in, enabling performance analysis or enabling a better distribution of responsabilities if an actor is identified as being overworked.

Link types of the Accountability link type enable the ability to assign authorship for actions. They connect artifacts to an actor which acted on them. Ideally, a traceability model should have link types which assign authorship to at least six of the seven basic actions described in Section 3.2; these actions are: creation, modification, removal, application, decomposition, and homologation. Reification is a special case of the creation action; thus, accountability for the reification action is established by the creation action; i.e., the actors who created a reified artifact are the actors who performed the action of reification.

3.3.2.4 Permission

Not every actor should be allowed to manipulate every artifact; manipulation of artifacts should be restricted according to roles and positions held by actors. For instance, a front-end developer should not be allowed to modify the same artifacts as a back-end developer. Restricting actions on artifacts according to roles, or individuals, is typical in software projects.

If a project does not restrict actors from manipulating artifacts which they should not manipulate, problems may arise by errors and maybe even by malicious reasons. To

record these restrictions is to trace them; i.e., if a project records this set of relations, it is generating traceability links of the Permission link type.

A project which does not have this set of relations is not able to restrict actors from manipulating artifacts which they should not manipulate. A project which has this set of relations, but uses a traceability model without Permission link types is not modeling useful relations; thus, they may not be considered during traceability related tasks, such as tasks performed by traceability processes. This may cause loss of traceability information and generation of consistency problems. For instance, it may be necessary to update permissions after a change (see Section 9.8 in Chapter 9).

Link types of the Permission link type convey information on authorization to perform actions on artifacts. It connects an artifact to an actor which has permission to perform a certain action on it. Ideally, a traceability model should have link types which convey information on authorization to perform at least six of the seven basic actions described in Section 3.2; these actions are: creation, modification, removal, application, decomposition, and homologation. Missing one of these six basic actions means not tracing an existing relation and not restricting, or restricting, actors from performing this action. Analogously to the previous link type, permission for the reification action is established by the creation action.

3.3.2.5 Characterize Action

Traceability enables the identification of rationale for actions. This may occur in many ways, from simpler ones such as a comment in source code, to more detailed ones such as a document explaining why a modification took place. To avoid losing this rationale it is necessary to have links modeling the relations between the artifact which went through the action and the artifact which justifies the action.

Sometimes a rationale is not enough; it may be desirable to have a rationale for the action and also a description of how the action should be done. For instance, an artifact may justify why a modification is necessary and describe how the modification should be performed; there may be essential steps to be done during the modification to avoid breaking some other functionality, or the modification should be done in a certain way to ensure it is done properly. On the other hand, only a description of an action may be necessary; for instance, a test case describing how it should applied on a code fragment. There are three possible configurations: (i) justifying and describing an action; (ii) solely justifying an action; and (iii) solely describing an action. The first provides rationale and describes an action; the second only justifies an action without providing instructions on how to perform the action; the third describes the action to be performed.

A traceability model lacking Characterize Action link types can not connect rationales and/or descriptions for actions performed on artifacts. Hence, a project using this traceability model does not have the ability to justify and/or describe actions; if the project has rationales for actions, the model fails at modeling existing relations.

Link types of the Characterize Action link type convey the rationale and/or description (what and how) concerning an action. It connects the artifact acted on to the rationale/description artifact for the action. Ideally, a traceability model should have link

types which convey the rationale and/or description to perform an action for at least the seven basic actions described in Section 3.2.

3.3.2.6 Action Outcome

Sometimes artifacts are created by actions performed on other artifacts. These artifacts are usually strongly related to the artifact which caused the action and/or the artifact acted on. For instance, a set of test cases applied on a program produces a test log containing the results of the test; this resulting artifact relates to both the program and the test cases. A traceability model may choose to connect the resulting artifact to the artifact which caused the action, to the artifact acted on, or to both. In these three options, traceability information is not lost, and the resulting artifact may be easily found.

A traceability model lacking Action Outcome link types can not connect artifacts created by actions to its closely related artifacts. Hence, a project using this traceability model does not have the ability to identify which artifact is the result of the action and which artifact caused the action (or was acted on). This traceability model is missing a typical relation between artifacts.

Link types of the Action Outcome link type convey the outcome of an action. It connects the resulting artifact to the artifact which caused the action and/or the artifact acted on.

3.3.2.7 Composition

During development it may be necessary to break artifacts into smaller parts. As mentioned in Section 3.2, this is a common action in software development which may be done, for instance, to increase cohesion. It may be interesting to keep the decomposed artifact for historical reasons, to enable the possibility of redoing its decomposition in a different way, among other reasons. To be able to identify this artifact it is necessary to have traceability links connecting it to its decomposed parts.

Traceability enables the possibility of tracing information having any size, or form, since an artifact contains information which may be divided into smaller pieces of information. To enable identification of parts of a working artifact without decomposing it, a link type which models the relation connecting the whole artifact to its parts is necessary. For instance, if an artifact is a class and another artifact is a method of this class, it is not possible to know from which class the method artifact is part of without a link type modeling the relation between them.

A traceability model lacking Composition link types can not connect decomposed artifacts to its decomposed parts. Hence, a project using this traceability model does not have the ability to identify the original artifact from which an artifact was decomposed from and vice versa.

A traceability model lacking Composition link types also can not connect an artifact to its parts. Hence, a project using this traceability model does not have the ability to identify if an artifact is part of another artifact or which artifact a “part” artifact belongs to.

Link types of the Composition link type convey the information that an artifact was, or is, part of another artifact; i.e., these link types convey that: (i) an artifact was decomposed into two or more artifacts, or (ii) an artifact is part of another artifact. It connects an artifact to the artifact from which it was decomposed from or connects an artifact to one of its parts.

3.3.3 About the Basic Link Types

These link types are proposed following our investigation on the subject of traceability in software development; there may be relations which are not modeled by this set of link types. Also, there is subjectivity involved when defining if a relation is basic "enough" to be represented in the Reference Model.

We do not argue that this set of basic link types is complete or final. It is possible that there are other basic relations which we did not consider and that can be added in the future. On the other hand, we consider these link types to be necessary, and extensive enough, for most software projects.

There may be specific contexts which demand additional link types. For instance, new link types may be necessary for security focused projects. However, the proposed link types are still valid for these specific contexts, since, as argued, they correspond to basic relations.

3.3.4 Justifying the Relations Modeled by the Reference Model

When creating a Reference Model, it must be decided which relations should be represented by link types and which relations do not need representation. A Reference Model should be useful; thus, the attribute being considered is usefulness. Usefulness depends on context; a Reference Model for a specific domain is different from a Reference Model for general use.

This Reference Model was created with the goal of being useful for general use; i.e., common development projects. When considering general use, the metric of usefulness is directly related to how common the relations being modeled by the Reference Model are. A relation should be modeled if it occurs commonly in projects of software development. Also, a Reference Model which models a common relation has more usefulness than a Reference Model which does not, even if considering specific domains. A specific domain Reference Model will probably use common relations; e.g., a Reference Model for traceability in safety-focused development also uses the dependency relation.

The proposed link types are common enough, modeling know relations between elements, and modeling relations which occur in typical software development projects. This is discussed next, where each modeled relation is evaluated on being typical enough to belong to this Reference Model. We also briefly explain how we identified each link type.

3.3.4.1 Evolution

Information is propagated during software development. The creation of new artifacts, in most cases, happens from the propagation of information. We identified two types of

propagation of information, given two of the basic actions we have identified previously: modification and reification. In the former, information is propagated and updated; in the latter, information becomes reified.

Being a common occurrence in software development, and by associating to two of the basic actions, we consider this relation to be common enough to be represented by a link type in the Reference Model.

3.3.4.2 Constraint

Many of the papers found [5, 67, 44, 23, 52, 53, 35] highlight two distinct constraints between artifacts: dependency and conflict. In the former, an artifact requires another artifact; in the latter, it is the opposite situation, where an artifact requires the non-existence, or absence, or another artifact. These are common relations, and are not exclusive to software development.

3.3.4.3 Accountability

The literature review revealed that the vast majority of models do not consider actors interacting in the project. Consequently, there is no modeling of the relations between actors and artifacts. To solve this issue, we have evaluated the possible relations between actors and artifacts in a project. Given our conceptual model, actors manipulate artifacts and perform processes. This manipulation of artifacts is done by performing actions. Thus, there is a relation of accountability between the actor who performs an action and the artifact in which the action is performed; i.e., the relation identifies who performed the action. Every project has actors manipulating artifacts; consequently, this is a common relation.

3.3.4.4 Permission

After identifying the accountability relation, we identified another relation concerning actors and artifacts: the relation of authorization regarding manipulation of artifacts. In software development, and in many other areas of development, determining who may perform an action is essential. This is essential for security reasons or to prevent personnel from accessing artifacts which they are not trained to manipulate. Not restricting access to certain actors may cause security issues or the generation of severe errors.

This is a typical relation between actors and artifacts in development projects; a software tester usually do not have authorization to modify code, for instance.

3.3.4.5 Characterize Action

One of the benefits of traceability is that it enables the possibility of keeping rationale for decisions made during development. Many of the papers found focus on traceability and design decisions [72, 8, 69, 13, 44].

Each action performed in the project has a rationale behind it; thus, the relation between the rationale and the subject of an action is common in projects. The Characterize Action link type models this relation.

3.3.4.6 Action Outcome

Another issue found in the literature review was that the activities of verification & validation and testing are rarely considered in traceability-themed papers. Testing is a common activity in software development and, when performed, results in the creation of artifacts; e.g., test logs. It is not possible to trace these new artifacts without a link type modeling the relation between generated elements and the elements which generated them. The Action Outcome link type models this relation and other possible relations in which an artifact creates another artifact as a result of an action being performed. Given how typical software testing is, we consider this relation as common.

3.3.4.7 Composition

There are two types of composition relations: the part-of relation and the decomposition relation. The part-of relation is a very common relation in software development; e.g., the composition relationship in the class diagram of UML.

The concept of decomposition of requirements is ubiquitous in the literature [15, 36, 37, 1, 30]; also, decomposition is a basic action, commonly occurring when developing software. This action is not restricted to requirements, being useful for other types of artifacts; e.g. decomposing code artifacts to increase cohesion. In conclusion, this is a typical relation.

3.4 Artifact Types

Artifact types model different types of products created during the development process. These products are represented by artifacts, which are instances of artifact types.

Unlike link types, the basic set of artifact types may change drastically from a development process to another. A classical development process may produce requirements, design documents, test cases, and other products; an agile development process may not produce, for instance, design documents. Given the intrinsic relation between the process used and the set of created products, it is difficult to identify basic artifact types which are common to all development projects. An ample set of basic artifact types is able to model different activities in software development but may have unused artifacts for certain processes; a minimal set of basic artifact types may not have unused artifacts but may fail to model typical artifacts. Hence, it is more desirable to create a set which models the most typical artifacts in software development.

To define the set of basic artifact types we used the most common activities in software development described in the Software Engineering Body of Knowledge [62]. Certain development processes, such as agile processes, may not have all of these activities; in this case, the user of the Reference Model just needs to discard the artifact types modeling the unused activities. These artifact types are generic enough to contain more specific artifacts used in most development processes.

3.4.1 Pre-Requirements

Artifact types within this category are created in the activities preceding the requirements engineering activity. Documents describing organizational needs and system objectives are examples of artifacts in this category.

3.4.2 Requirements

Artifact types within this category are created in the requirements engineering activity. This activity being defined as “the process of defining, documenting and maintaining requirements” [38]. Requirements specifications and requirements elicitation documents are examples of artifacts in this category.

3.4.3 Design

Artifact types within this category are created in the design activity. This activity being defined as “the process of defining the architecture, components, interfaces, and other characteristics of a system or component” [62]. UML diagrams and documents detailing user interfaces are examples of artifacts in this category.

3.4.4 Implementation

Artifact types within this category are created in the implementation activity. This activity being defined as the creation of software through coding and the generation of related elements. Source-code, code documentation and instruction manuals are examples of artifacts in this category.

3.4.5 Verification & Validation and Testing

Artifact types within this category are created for the activities of verification & validation and testing. Verification & validation determines if a product conforms to its set of specifications and whether it fulfills its intended purpose; testing is defined as “the dynamic verification that a program provides expected behaviors on a finite set of test cases, suitably selected from the usually infinite execution domain” [62]. A document used to verify & validate requirements and a set of, or individual, test cases are examples of artifacts in this category.

3.4.6 Actors Are Not Artifacts But...

Actors are not bonafide artifacts, however, for traceability purposes, they may be treated as such. Hence, they deserve to be listed as a – special kind of – artifact type in a traceability model.

An actor may be a digital representation of a person (the exception are automated systems interacting with the project), therefore it has an artifact-like representation in a project using a traceability model. An actor modeling a developer will have all relevant information about this person; for instance: identifiable number, personal data, and so on.

Changes in the project may cause changes in the actor "artifact", adding new traceability information.

3.5 Ensuring Traceability and System Consistency

It is not enough for a traceability model to provide link types and artifact types. A project with integrated traceability may become inconsistent, and have wrong traceability information, if it goes through changes. Erroneous traceability information can not be used to ensure system consistency, one of the main benefits of having traceability. For instance, suppose an artifact, which is necessary for another artifact, is modified and this modification nulls its capacity to solve the dependency, leaving the system with another artifact having an unfulfilled dependency. If the relevant links are not updated accordingly, it may not be possible to identify, and consequently solve the issue. Hence, a traceability model must be able to ensure *traceability consistency* to ensure *system consistency.*

Definition 3.5.1. Traceability consistency is the property by which all links in a project are always up to date despite changes. All traceability information must be in sync to the current system state; for instance, if an artifact β is created to replace a previous version α, the traceability links arriving in α should be redirected to β [2] . Traceability consistency is an essential property to ensure system consistency.

Definition 3.5.2. System consistency is the property by which elements of a project are coherent among themselves. For instance, if a system has an unfulfilled dependency, it is not consistent. A traceability model must ensure system consistency. If a model ensures only traceability consistency it is not using the updated traceability information to keep the project working correctly.

Traceability processes are the mechanisms used to ensure traceability consistency and system consistency.

3.5.1 Basic Processes

The goal of a traceability process is to maintain traceability consistency and system consistency. Consistency may be broken by actions done throughout the development process. For instance, suppose an artifact α is necessary for another artifact β; α undergoes a modification which removes its capacity to fulfill the dependency of β. In this example, both the system, and the traceability information of the system, became inconsistent; the artifact β has an unfulfilled dependency and there is a link representing a relation which does not exist anymore.

Since consistency is broken by actions performed in the project, a traceability model should have processes for, at least, all the basic actions on artifacts.

Additional processes may be needed given specific contexts, but this set of processes is also useful for specific contexts since they correspond to basic actions.

[2]Not necessarily all links.

3.6 Necessary Properties of Link Types and Artifact Types

When discovering shortcomings in the traceability models found in the literature, we established three basic properties which link types and artifact types should have. All papers lacked these properties to some extent; most papers lacked all three properties, and some papers lacked two of these properties. The three necessary properties of link types and artifact types are: Comprehensiveness, Specificity, and Coverage.

3.6.1 Comprehensiveness

Comprehensiveness is the property of taking into account a wide range of situations when modeling an element. A set of link types having the property of comprehensiveness is able to model the most common relations between artifacts and artifacts, and artifacts and actors in a software development project. Analogously, a set of artifact types having comprehensiveness is able to model the most common artifacts in a project. For instance, consider a set composed of three link types, L_1, L_2 and L_3, modeling three different relations: permission, conflict, and accountability, respectively. This set is unable to model a relation of dependency between two artifacts; it is not comprehensive enough. A set of link types lacking comprehensiveness fails to cover certain relations. This property also applies to sets of artifact types. A set of artifact types which does not have a type to model code elements may not be comprehensive enough for software development.

For link types and artifact types this property may be fulfilled by creating very general types. For instance, a set may define that a "Trace" link type models every single possible relation. However, this leads the traceability model to lack the next property, specificity.

3.6.2 Specificity

Specificity is the property of being capable to model specific elements. Consider the previous example of a set having the Trace link type; this set has the property of comprehensiveness, to perfection, since it represents every existing relation. However, this set fails to characterize any specific relation, resulting in losing too much traceability information and consequently rendering this set useless for traceability purposes. For instance, a traceability link of the Trace link type would model both a dependency relation (Constraint link type) and an accountability relation (Accountability link type), but we are unable to identify which one is being modeled. Hence, a set having comprehensiveness but lacking specificity is able to model a diversity of relations, or artifacts, but loses traceability information.

Ideally, a set should have comprehensiveness by modeling a substantial number of relations by different types, instead of using types which are too generic; i.e., creating a set having too much specificity may cause it to lose comprehensiveness if there is not enough distinct link types modeling the relations.

The property of specificity is more relevant for link types than for artifact types; link types are the primary providers of traceability while artifacts add to the traceability infor-

mation. Having high specificity artifact types may increase the traceability information in a project, but not to the same extent as specific link types. Artifact types add information to the semantics of link types since link types may have more detailed descriptions taking into account which artifact types they connect; artifact types are also used in traceability processes.

3.6.3 Artifact Coverage

Artifact Coverage is the property by which each link type in a set covers all the artifact types they should cover. A set of link types having artifact coverage is able to represent the necessary relations for every tuple of artifacts.

This property is lacking in most graphical models found in the literature, in which the model has rigid relations between artifacts. For instance, consider a traceability model having a link type which expresses authorship on the creation of artifacts; this link type connects actors to two types of artifacts, but does not connect actors to a third type; i.e., the author of the third artifact is not being traced. Since it is desirable to have the knowledge of the author of each created artifact, this traceability model lacks artifact coverage.

3.6.4 About the Necessary Properties

There is a degree of subjectivity when evaluating all three properties, which probably can not be avoided. Consequently, these are guidelines to be taken into consideration when creating sets of elements in a traceability model.

Domain and context should be considered during the evaluation; not every link type is necessary for every domain and context. On the other hand, certain link types may be absolutely necessary in certain domains and contexts.

These properties are also useful to evaluate current traceability models; they enable assessing how complete is a set of link types or artifact types of a model, given a certain domain and context. For instance, a traceability model for general software development which does not have link types to represent dependency and accountability relations will be considered incomplete in many contexts.

Chapter 4

Using the Reference Model

The Reference Model may be used to create traceability models or to evaluate traceability models. A model may be evaluated given the basic properties and elements defined by the Reference Model; for instance, by mapping the link types of a model to the basic link types of the Reference Model, we are able to assess if there are basic relations not being considered by the model.

In this chapter, we use the Reference Model to evaluate two relevant contributions in the literature: a paper by Ramesh and Jarke [53] and a set of papers by Goknil et al. [23, 21, 22, 24, 25]; these are the papers we consider most closely related to our research.

The contribution by Ramesh and Jarke is an empirical work cited by many papers in the literature; the contribution by Goknil et al. is composed of papers containing a traceability model applied to different subjects while going through a few changes.

This chapter is structured as follows: Section 4.1 specifies the strategy used to perform the evaluation; Section 4.2 shows the evaluation of the contribution by Ramesh and Jarke; Section 4.3 shows the evaluation of the contribution by Goknil et al.; and Section 4.4 provides a few closing remarks on the evaluations.

4.1 Evaluation Strategy

The evaluation takes into account the three basic elements in a traceability model: link types, artifact types, and processes.

First, the link types and artifact types are mapped into the types in the Reference Model. Then, the link types and artifact types are evaluated on: (i) description level; i.e., how well each type is described – we use the classification defined in Section 2.4.2: None (name only), Minimal (usually a sentence containing its name), Informal (informal description), or Formal (well-defined traceability relation, described formally); (ii) the three basic properties of comprehensiveness, specificity, and coverage – these properties are detailed in Section 3.6.

The processes are evaluated on: (i) covering all basic actions; and (ii) capacity to ensure consistency – i.e., how well a process ensures consistency, identification of visible deficiencies.

Lastly, strengths and/or limitations of the contributions are discussed.

4.2 Ramesh and Jarke

Ramesh and Jarke [53] propose a traceability model grounded in empirical work in "Toward Reference Models for requirements traceability" from 2001; while being empirical, this paper was influenced by three previous papers from one of the authors:

- Issues in the development of a requirements traceability model (1993) [52];
- Towards requirements traceability models (1995) [51];
- Implementing requirements traceability: a case study (1995) [54];

these papers are discussed in Chapter 2.

The paper resulted from an empirical work; interviews were done in 26 software development companies. Two traceability models are proposed: a simple model (Low-End Model) and a detailed model (High-End Model). These are meant to be used according to each organization traceability needs. The Low-End Model contains the following link types: derive, developed for, allocated to, satisfy, performed on, depend on, and interface with. The Low-End Model contains the following artifact types: requirements, compliance verification procedures, system/subsystems/components, and external systems.

The High-End Model consists of four submodels: Requirements Management Submodel, Rationale Submodel, Design Allocation Submodel, and Compliance Verification Submodel.

The Requirements Management Submodel contains the following link types: derive, describe/describes, justify, identify, generate/generates, managed by, modify, elaborate, based on, part of, and depend on. The Requirements Management Submodel contains the following artifact types: organizational needs, operational needs, strategic needs, scenarios, system objectives, critical success factors, change proposal, resource, requirements, constraints and mandates; this last artifact is derived into standards, policies, and methods.

The Rationale Submodel contains the following link types: traces to, derive, based on, affect, generate, depend on/depends on, resolve, influence, evaluate, select, address, oppose, and support. The Rationale Submodel contains the following artifact types: object, components, requirements, designs, rationale, decisions, issues or conflicts, assumptions, critical success factors, alternatives, and arguments.

The Design Allocation Submodel contains the following link types: address, perform, depend on/depends on, part of, allocated to, satisfy, derive, create, define, used by, drive, modify, and based on. The Design Allocation Submodel contains the following artifact types: functions, external systems, requirements, system/subsystems/components, resources, design, change proposals, and mandates; this last artifact is derived into standards, policies, and methods.

The Compliance Verification Submodel contains the following link types: used by, generate, derive, based on, verified by, developed for, and satisfy. The Compliance Verification

Submodel contains the following artifact types: resources, change proposals, compliance verification procedures, inspection, test, prototype, simulation, system/subsystems/components, requirements, and mandates; this last artifact is derived into standards, policies, and methods.

Also, four general link types are proposed: satisfaction, evolution, rationale, and dependency; each link type from the submodels corresponds to one or more of these general types. However, the authors do not provide a complete classification of the specific link types; only a few examples are shown.

4.2.1 Classification of the Link Types

In Table 4.1, the general link types plus the specific examples shown in the paper are classified into the link types of the Reference Model. The first and second columns show the general link type and the related specific link type, from Ramesh and Jarke, respectively; the third column shows the corresponding link type from the Reference Model.

Table 4.1: Classification of the Link Types - Ramesh and Jarke.

Ramesh and Jarke Link Type		Reference Model Link Type
General Link Type	**Specific Link Type**	
Satisfaction	drive, allocated, satisfy, perform, address, developed for, verified by	Evolution
Evolution	generate, justify, describe, modify, identify, based on, derive, elaborate, affect, drive, define, create	Evolution
Rationale	based on, generate, address, support, oppose, depend on, select, evaluate, resolve, influence, managed by, use	Characterize Action
Dependency	depend on is a, part of	Constraint Composition

The Satisfaction and Evolution link types from Ramesh and Jarke are classified as Evolution; these represent reifications and modifications, both subtypes of the Evolution link type of the Reference Model. Rationale conveys rationale; thus, it is classified as Characterize Action. Dependency is classified as Constraint, since it represents the dependency subtype of the Constraint link type.

4.2.2 Description Level of the Link Types

The general link types are semantically described. There is no formalization of their description. Hence, their description level is informal.

The specific link types are not semantically described, and consequently there is no formalization of their description. Hence, their description level is minimal; all link types

are described in a sentence containing artifacts and using the name of the link type as a verb.

4.2.3 Evaluation of the Necessary Properties of Link Types

The set of link types lacks comprehensiveness and coverage. The set of link types may, or may not, have specificity, depending on which set of link types is evaluated.

The set of link types lacks comprehensiveness; it fulfills only four of the seven basic link types of the Reference Model. Moreover, these four basic link types are not completely fulfilled. For instance, the Constraint link type contains two subtypes, conflict and dependency; the model does not model conflict[1].

The link types seem to have specificity; it is not trivial to deduce this information given the minimal description provided for the specific link types. If we consider only the four general link types, the set does not have specificity.

The set of link types lacks coverage. Here are some of the many issues found: the depend on and depends on link types do not cover design artifacts; hence, the model does not enable modeling dependency relations between design artifacts. The modify link type covers only design and requirement artifacts, precluding tracing the modification of other artifact types. The part-of link type models only the relation between system, subsystem, and requirement artifact types, not covering design artifacts, for instance; thus, it precludes tracing composition relations between design artifacts.

4.2.4 Classification of the Artifact Types

The artifact types shown in the paper, organized by submodel, are classified into the artifact types from the Reference Model in Tables 4.2–4.5. The first column shows the artifact type from Ramesh and Jarke, and the second column shows the corresponding artifact from the Reference Model.

Table 4.2: Classification of the Artifact Types - Ramesh and Jarke - Requirements Management Submodel

Ramesh and Jarke Artifact Type	Reference Model Artifact Type
organizational needs, operational needs, strategic needs, scenarios, system objectives, critical success factors, resource, mandates	Pre-Requirements
requirements, constraints, change proposal	Requirements

In the Requirements Management Submodel, the *change proposals* artifact modifies only *requirements* artifacts; hence, it is classified as a Requirement artifact type from the Reference Model.

[1] Conflict is only modeled as an artifact; for instance, a list of conflicts.

Table 4.3: Classification of the Artifact Types - Ramesh and Jarke - Rationale Submodel

Ramesh and Jarke Artifact Type	Reference Model Artifact Type
requirements	Requirements
designs	Design
components	Implementation
decisions, issues or conflicts, assumptions, critical success factors, alternatives, arguments	Requirements, Implementation, Design
rationale	Rationale

The artifact type *object* from the Rationale Submodel was not classified since it is a meta-type from which all other artifact types derive.

The artifact types *rationale, decisions, issues or conflicts, assumptions, critical success factors, alternatives,* and *arguments* occur in three distinct development activities, hence, they are classified as three artifact types from the Reference Model.

Table 4.4: Classification of the Artifact Types - Ramesh and Jarke - Design Allocation Submodel

Ramesh and Jarke Artifact Type	Reference Model Artifact Type
mandates	Pre-Requirements
requirements	Requirements
design, change proposals	Design
functions, external systems, system/subsystems/components	Implementation

In the Design Allocation Submodel, the *change proposals* artifact modifies only *design* artifacts; hence, it is classified as a Design artifact type from the Reference Model.

The artifact type *resources* is not classified since it models elements which are external to the development and non-actor people; quote from the paper: "...money, weight, personnel, power...".

Table 4.5: Classification of the Artifact Types - Ramesh and Jarke - Compliance Verification Submodel

Ramesh and Jarke Artifact Type	Reference Model Artifact Type
mandates	Pre-Requirements
requirements	Requirements
system/subsystems/components	Implementation
test, compliance verification procedures, inspection, prototype, simulation	Verification & Validation and Testing
change proposals	Requirements, Implementation, Design

In the Compliance Verification Submodel, the *compliance verification procedures* artifact generates *change proposals* artifacts for requirements, design, and implementation; hence, it is classified as Requirements, Implementation, and Design artifact types from the Reference Model.

4.2.5 Description Level of the Artifact Types

The artifact types are described in sentences; sometimes there are examples to help explain the type. They are described, but not with detail. There is no formalization of their description. Hence, their description level is classified as informal.

4.2.6 Evaluation of the Necessary Properties of Artifact Types

The set of artifact types from this work does not have comprehensiveness but have specificity.

The set of artifact types does not have (complete) comprehensiveness; it fulfills five of the six artifact types of the Reference Model but does not model actors interacting in the project. Since it does not model actors, there are no link types representing relations between actors and artifacts.

Given that the paper does not focus on a specific domain, the proposed artifact types are specific enough.

4.2.7 Processes

The authors do not propose processes to ensure Traceability Consistency and System Consistency.

4.2.8 Strengths & Limitations

This contribution has a strong empirical basis; thus, it provides practical perceptions of traceability. It also has artifacts modeling the most common activities in software development, which it not true for the majority of studied papers. Finally, it contains a rationale submodel, enabling the tracing of reasons for actions performed in the project.

On the other hand, there are several limitations. It models the most common activities of development but does not models actors; consequently it is not capable of modeling any relation between actors and artifacts. The descriptions of the specific link types is minimal; it does not describe each link type. There are four general link types which are described, but the specific link types are not classified into these more general types. Therefore, when utilizing this model, a user has to deduce the semantics of each specific link types to map these to relations in a project. For instance, design *creates* or *defines* systems/subsystems/components; what is the difference when design defines a component versus when design creates a component? That is, what are the semantic differences between these two link types?

The link types also lack comprehensiveness and coverage; this restricts the traceability information acquired when using the model.

No processes are provided; hence, it can not ensure traceability and system consistency whenever a change occurs in a project using this model.

4.3 Goknil et al.

Goknil et al. [23, 21, 22, 24, 25] contribution is composed of five distinct papers, all of which use the same traceability model to achieve different goals. This model is proposed in [23] and goes through improvements and changes from one work to another. Not all papers use the complete model. The five papers are:

- A Metamodeling Approach for Reasoning about Requirements (2008) [23];
- Semantics of trace relations in requirements models for consistency checking and inferencing (2011) [21];
- A metamodeling approach for reasoning on multiple requirements models (2013) [22];
- Generation and validation of traces between requirements and architecture based on formal trace semantics (2014) [24];
- Change impact analysis for requirements: A metamodeling approach (2014) [25].

"A Metamodeling Approach for Reasoning about Requirements" [23] proposes a model for requirements and a strategy to customize the model to support different techniques for requirement specification. Four link types are proposed: Requires, Refines, Contains, and Conflicts. These are formalized in first-order logic. An industrial mobile service application is used in a case study.

"Semantics of Trace Relations in Requirements Models for Consistency Checking and Inferencing" [21] proposes a method for consistency checking and inference of new relations, given the model proposed in [23]. TRIC [68] is a tool developed to support these activities. A new link type – PartiallyRefines – is added to the model; this link type is a variation of the Refines link type.

"A Metamodeling Approach for Reasoning on Multiple Requirements Models" [22] expands the contribution from 2008 by adding support to different approaches of requirements modeling. A new link type – Equals – is added to the model; this link type identifies a copy of a requirement. This was probably added to enable mapping the model to SysML. A more useful link type would identify equal parts of requirements, instead of whole requirements. This link type was removed from all subsequent works using the same model.

"Generation and Validation of Traces Between Requirements and Architecture Based on Formal Trace Semantics" [24] proposes an approach to automatically generate traces between requirements and architecture; the relations defined in previous works are used to achieve this goal. Two new link types are added to the model, to support tracing between requirements and architecture: AllocatedTo and Satisfies. These were taken from the contribution by Ramesh and Jarke [53]. A tool based on the Eclipse Modeling

Framework, ATL modeling transformation language and the Maude tool set, is proposed to support the generation and validation of traces.

The paper "Change impact analysis for requirements: A metamodeling approach" [25] provides a formal classification of changes in requirements; the formal definitions of relations and the formal definition of types of changes enable: identification of alternative changes, identification of incorrect positive impacts, and consistency checking of changes. This is used to create a change impact analysis approach for requirements; TRIC [68], proposed in [21], is expanded to support this new approach.

4.3.1 Classification of the Link Types

The link types are classified into the link types from the Reference Model in Table 4.6. The first columns shows the link type from Goknil et al. contribution; the second column shows the corresponding link type from the Reference Model.

Table 4.6: Classification of the Link Types - Goknil et al.

Goknil et al. Link Type	Reference Model Link Type
requires, conflicts	Constraint
refines, partially refines, allocatedTo, satisfies	Evolution
contains	Composition

The requires link type and the conflicts link type are classified as Constraint; they model the dependency relation and the conflict relation of the Constraint link type, respectively. The link types refines and partially refines are classified as Evolution; they model a refinement between requirements. The link types allocatedTo and satisfies are classified as Evolution; they model refinements between elements of the requirements model and the architectural model. The difference between the allocatedTo link type and the satisfies link types is how they are assigned: the former is assigned automatically and the latter is assigned manually by an actor. The contains link type is classified as Composition; it refers to requirements which are part of other requirements (relation whole/part of).

4.3.2 Description Level of the Link Types

The link types are semantically described and are formalized in first-order logic. Hence, their description level is formal.

4.3.3 Evaluation of the Necessary Properties of Link Types

The set of link types lacks comprehensiveness; it fulfills only three of the seven basic link types of the Reference Model. Moreover, these three basic link types are not completely fulfilled, since the Evolution link type has two subtypes and only one is fulfilled; the model does not model modification.

The set of link types has specificity, being specific enough for general use in software development. For example, consider the refines link type: it models a reification relation but does not model the modification relation (both are subtypes of the Evolution link type).

The set of link types have coverage, since there is only one artifact to cover; all link types model relations between requirements.

4.3.4 Classification of the Artifact Types

The proposed link types model only relations between requirements. In two of the papers, they provide a model containing other types of artifacts such as StakeHolder and TestCase; however, these are not considered for traceability. One of the papers considers an architectural view of development, but no specific elements are provided. Therefore, there is only one artifact type to be classified.

Table 4.7 shows the classification. The first column shows the artifact type from Goknil et al., and the second column shows the corresponding artifact from the Reference Model.

Table 4.7: Classification of the Artifact Types - Goknil et al.

Goknil et al. Artifact Type	Reference Model Artifact Type
requirements	Requirements

If we consider elements which are not traced, Stakeholder is classified as Actor and TestCase is classified as Verification & Validation and Testing.

4.3.5 Description Level of the Artifact Types

There is only one artifact type: requirements; it is semantically described and formalized. Hence, its description level is formal.

4.3.6 Evaluation of the Necessary Properties of Artifact Types

The set of artifact types lacks comprehensiveness; it fulfills only one of the six basic artifact types of the Reference Model.

The set of artifact types has specificity; it models an element which is used in practical development (requirement).

4.3.7 Processes

The authors do not propose processes to ensure Traceability Consistency and System Consistency. However, the authors provide an algorithm in [25] to evaluate the impact of changes caused by the modification of an requirement. It was not created to ensure consistency but could be adapted accordingly; if such adaptation was done, the resulting process would have some flaws: it would not take into consideration system breaking

issues such as unsatisfied dependencies, or conflicts, between elements, given a certain change; it would not consider different actions, such as the decomposition of an element; also, the limited types of relations being considered would weaken the algorithm, since it would not consider relations such as those between elements and rationales for actions, or elements and byproducts of their application; among other issues.

4.3.8 Strengths & Limitations

The link types are defined formally in first-order logic. The requirement artifact type is also defined formally. This contribution is superior to all other works investigated in Chapter 2 when taking into account the quality of descriptions of elements. Every link type and artifact type is well-defined; thus, there would be no ambiguity when using the proposed types in a project.

On the other hand, this model is limited to requirements, not considering all other activities in software development. It also does not model actors which interact with the project. While having specificity, the set of link types lacks in comprehensiveness even more than the contribution by Ramesh and Jarke [53]; e.g., using this set of link types it is not possible to trace reasons for actions done in the project. It does not propose processes to ensure consistency; however, it is the only contribution studied which contains an algorithm which could be adapted to become a process. Still, this process would have flaws and would only cover, at most, two of the basic actions (modification and reification).

4.4 Closing Remarks

This evaluation intends to illustrate the use of the Reference Model as an evaluating tool. During this evaluation, we have gained knowledge about common flaws of traceability models and improved our understanding of traceability.

We consider these to be the strongest contributions in the literature review. The work by Ramesh and Jarke considers the most common development activities and contains many link types; however, it does not provide proper descriptions of the proposed link types. The contribution by Goknil et al. contains formally defined elements; on the other hand, it considers only the requirement engineering activity and proposes few link types.

Part III

A Metamodel for Traceability

Chapter 5

An Overview of the Metamodel

The Reference Model described in Chapter 3 defines the basic elements in a traceability model: artifact types, link types, processes, and actors. Artifact types model products from development activities. Link types model relations between the artifacts. Processes ensure consistency given changes, caused by actors, on the artifacts. These four elements are intrinsically related to actions which occur in a development project; e.g., if an action of modification is performed by an actor on an artifact, a link type may provide accountability for this action, and a process may be needed to ensure the project does not become inconsistent.

The Reference Model determines sets of: (i) actions, (ii) link types, (iii) artifact types, and (iv) processes. Each set is basic; i.e., contains the minimum necessary elements for general use. Therefore, the Reference Model answers the following questions about traceability models: which actions should be considered? Which link types and artifact types are necessary? Which properties these sets of types should have? Which processes are necessary? As mentioned previously, all these elements relate strongly to each other; for instance, the set of basic actions influences the elements of the set of types and the set of processes. If modification is an action in the set of basic actions, there should be: accountability, permission, characterize action and evolution link types covering this action, and a process which ensures consistency whenever this action is performed. Each set was defined, and in some cases, subtypes were also determined as necessary; e.g., the accountability link type should cover six of the seven basic actions.

In the following chapters, we present a Metamodel for traceability constructed by using the Reference Model from Chapter 3; the Metamodel is presented by detailing its four main components: the Traceability Space, the Artifact Types, the Link Types, and the Processes.

5.1 Traceability Space

The Traceability Space is the conceptual visualization of the Metamodel; it describes and organizes the elements of traceability. The Traceability Space contemplates the seven basic actions of the Reference Model plus the Activation action, which enables bringing unused artifacts into the working project.

5.2 Artifact Types

This is the set of Artifact Types of the Metamodel. As discussed in Chapter 3, a traceability model should define a set of Artifacts Types representing the products of the development process. Artifacts – instances of Artifact Types – are manipulated by Actors, handled by Processes, and traced by Traceability Links.

The Metamodel contains a set of 12 Artifact Types which represent the most common activities of development and record valuable traceability information.

5.3 Link Types

This is the set of Link Types of the Metamodel. As discussed in Chapter 3, a traceability model should define a set of Link Types modeling the relations between the products of the development process. Links – instances of Link Types – trace Actors and Artifacts, and are handled by Processes.

The Metamodel contains a set of 59 Link Types describing different traceability relations, created by taking into account the three necessary properties determined in the Reference Model.

5.4 Processes

This is the set of Processes of the Metamodel. As discussed in Chapter 3, a traceability model should define a set of Processes to ensure consistency. Consistency is broken by actions; thus, a model should have a Process for each Action contemplated by the Metamodel. Processes are performed by Actors, and handle Artifacts and Links.

The Metamodel contains 8 traceability processes to enable the maintenance of traceability consistency and system consistency.

5.5 Outline by Chapter

The conceptual model and the actions contemplated by the Metamodel are defined in Chapter 6. The set of Artifact Types is defined in Chapter 7. The set of Link Types is defined in Chapter 8. The set of Processes is defined in Chapter 9. An application of the Metamodel on requirements of a course management system is shown in Chapter 10.

Chapter 6

The Traceability Space

The Traceability Space describes and organizes the elements of traceability. The Traceability Space of a project is composed of four subspaces: Actors Space, Rules Space, System Space and Processes Space. The Actors Space contains abstractions of the agents that interact with the project; the Rules Space contains rules limiting the actions which can be performed by the Actors; the System Space contains Artifacts (instances of Artifact types), which are the products of the development process; the Processes Space contains a collection of processes used to maintain consistency. Traceability links (instances of link types) are also part of the Traceability Space; they connect Artifacts to Artifacts within the System Space, and Artifacts in the System Space to actors in the Actors Space.

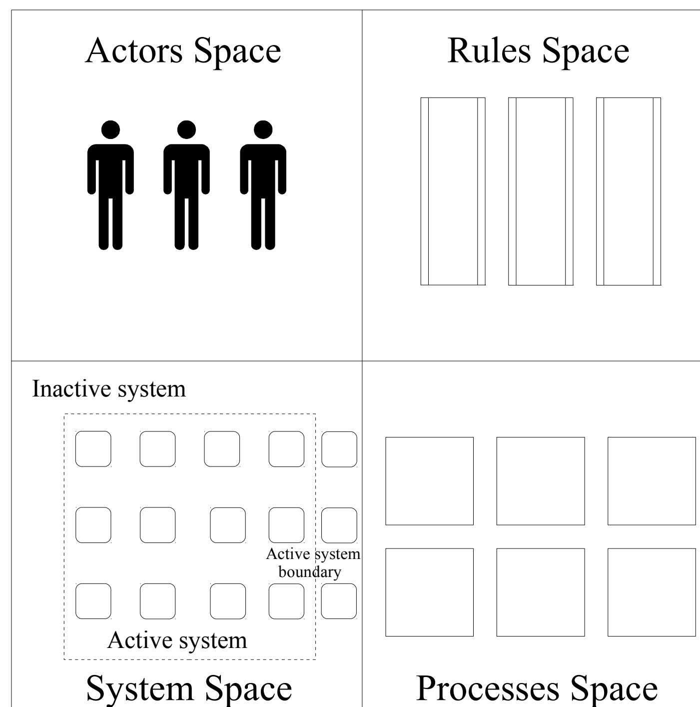

Figure 6.1: The Subspaces of the Traceability Space.

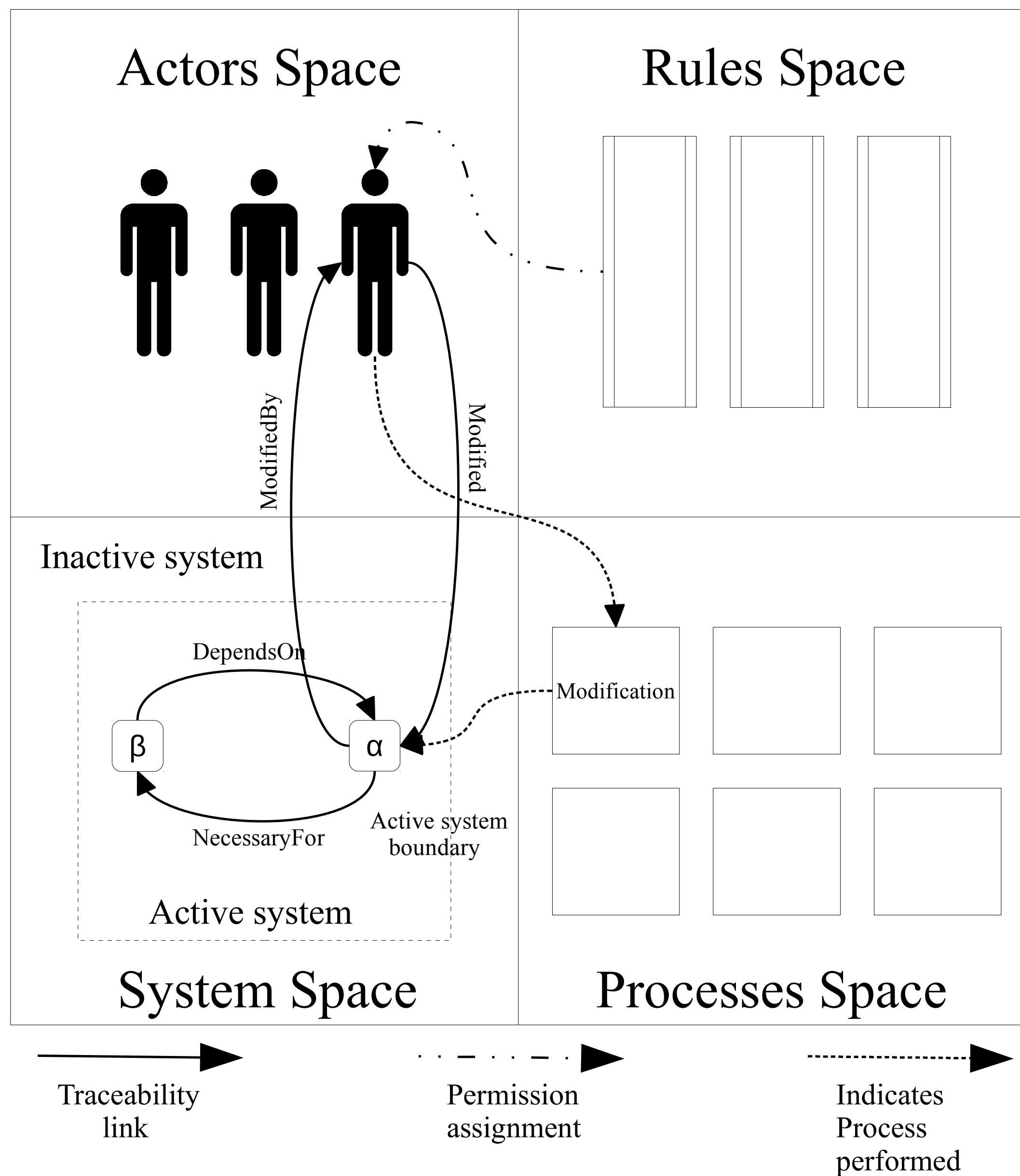

Figure 6.2: A Traceability Space.

A Traceability Space is depicted in Figure 6.2. An actor in the Actors Space, having permission given by the Rules Space, performed the Modification Process, from the Processes Space, on artifact α in the System Space. The resulting Accountability links *ModifiedBy* and *Modified* connect the actor and the artifact. The Constraint links *DependsOn* and *NecessaryFor* connect artifacts α and β.

This chapter is structured as follows: Section 6.1 details the interactions between the basic elements of traceability given this conceptual model; Sections 6.2–6.5 describe the elements of the Traceability Space; Section 6.6 defines the Traceability Space; Section 6.7 defines the basic actions used in the Metamodel; and Section 6.8 briefly discusses possible metrics enabled by Metamodel.

6.1 Interactions Between the Elements of Each Space

The Reference Model defines three basic elements of a traceability model: Artifacts, Links, and Processes; Actors are agents which interact with these elements.

The Artifacts in the System Space are instances of the Artifact types from the set of Artifact types of the Metamodel. These are manipulated by Actors in the Actors Space.

Actors perform actions on Artifacts in the System Space; these are the actions contemplated and defined by the Metamodel.

Actors are able to manipulate Artifacts if they have authorization defined by the Rules Space. This authorization may be fully, partially, or not at all, expressed by traceability links connecting the Actors to the Artifacts. For instance, a rule from the Rules Space may determine that a certain Actor may modify a specific Artifact, and another rule from the Rules Space may determine that another Actor may create Artifacts of a certain type during a specific time window. The former is expressed by traceability links; the latter is not expressed by traceability links.

Traceability links are instances of link types from the set of link types of the Metamodel; these connect: Actors in the Actors Space to Artifacts in the System Space, and Artifacts in the System Space to other Artifacts in the System Space.

The processes from the Processes Space contain actions to be performed and the instructions necessary to ensure traceability and system consistency. The Actors use the processes to enact changes in the System Space.

6.2 Actors Space

The Actors Space contains the actors participating or interacting with the project. Actors are digital representations of real life agents and can represent people or digital systems. Actors are connected to artifacts in the System Space by Traceability links; these links may be permissions or accountability for actions previously performed. Permissions give actors power to manipulate the artifacts in the System Space; e.g., a link which allows actor a to modify artifact α. Accountability enables identifying the actor's participation in actions performed on an artifact; e.g., a link which identifies that actor a took part in modificating artifact α.

The actor is a digital representation, hence it is able to store information. For instance, an actor in the Actor Space may have information on itself such as name, employee identifiable number, position held, etc.; all this information is stored in the actor representation in the Actors Space.

6.2.1 Meta-Actor

A meta-actor is an actor who is able to manipulate the Rules Space, the Processes Space, and the Actors Space by creating rules or assigning permissions manually, by creating, modifying, or removing processes, and by creating, modifying, or removing actors, respectively. An example would be a project manager who defines rules for the Rules Space of a project.

Traceability links may be used to trace accountability for the actions of the meta-actor in the subspaces of the Traceability Space.

6.3 Rules Space

The Rules Space contains rules concerning actions which may be performed in the System Space. The rules determine permissions which are assigned to each actor, establishing the set of allowed actions of an actor in the System Space.

The addition of new actors to the Actors Space or new artifacts to the System Space may bring the assignment of new permissions and may bring new rules. Permissions will be assigned to allow the new actor to manipulate current artifacts or to allow current actors to manipulate the new artifact. New rules may be created to adapt the Rules Space to the new actor or artifact; e.g., given a new artifact of a type not yet added to the System Space, a new rule would allow actors to manipulate artifacts of this specific type. Rules may be generated automatically or manually.

Rules can be as simple, or complex, as desired; each project establishes its set of rules according to its needs. For instance, a simple rule could be: actor a can modify artifact β. A more complex rule could be: actor a can modify and decompose all artifacts of type A_1 during a time window t. An even more complex rule could be: actor a can create artifacts of type A_1, under the condition that these can only be applied to artifacts of type A_2 created after a date d, during a time window t. Rules may also be generic, for instance: every new actor having position held ph can create and modify artifacts of type A_1.

6.4 Processes space

The Processes Space contains the processes necessary to maintain consistency. It contains a process for each action which may break traceability or system consistency. Whenever an actor performs an action, a process determines the required steps to be taken to ensure consistency. To satisfy the Reference Model, a Processes Space should have at least six processes covering six of the seven basic actions. In our Metamodel, there are seven processes: the Homologation Process, the Modification Process, the Decomposition Process, the Creation Process, the Removal Process, the Activation Process, and the Application Process.

The Homologation Process evaluates and enables artifacts to be added to the set of working artifacts of the project; an artifact which goes through the Homologation Process may be homologated or rejected.

The Modification Process evaluates and enables the modification of artifacts. An artifact which goes through the Modification Process is copied and this copy is subjected to the desired modification; after its modification, the new version takes the place of the original artifacts between the working artifacts of the project.

The Decomposition Process evaluates and enables artifacts to be decomposed into parts. The information from the artifact is kept in the created artifacts and no information

is added or removed. To add new information, the Modification Process should precede or succeed the Decomposition Process.

The Creation Process evaluates and enables the creation of new artifacts.

The Removal Process evaluates and enables the removal of artifacts from the set of working artifacts of the project. Artifacts are never destroyed, but instead are kept for future reuse, preserving rationale, or for historical purposes.

The Activation Process evaluates and enables artifacts, which were previously removed or rejected, to be added to the set of working artifacts.

The Application Process evaluates and enables artifacts to be applied on other artifacts. For instance, a test case is applied on a certain code fragment.

6.5 System Space

The System Space contains the products of the development process; i.e., it has all the artifacts produced during each phase of a given development process. It is separated into two parts: the Active System and the Inactive System. The Active System contains the artifacts currently being used in the project, the active artifacts. Examples of active artifacts are requirements which represent currently desirable, and approved, features and source code derived from these requirements. The Inactive System contains the artifacts not currently being used in the project, the inactive artifacts; these artifacts are stored for future reuse, rationale preservation, or for historical purposes. Examples of inactive artifacts are discarded requirements or previous versions of source code.

A new artifact starts in the Inactive System and, through the process of homologation, becomes part of the Active System; i.e., it starts off as an inactive artifact and may become an active artifact. The activation of an inactive artifact may happen by going through the Homologation Process directly or by going through other processes which use the Homologation Process to try to activate artifacts.

Currently inactive artifacts which were removed by the Removal Process or were previously rejected during the Homologation Process may become active by going through the Activation Process; this process uses the Homologation Process to activate artifacts.

An active artifact becomes inactive by going through the Removal Process, the Decomposition Process, the Modification Process, or by direct deactivation. In the first case, an artifact is simply removed from the Active System; in the second case, a decomposed artifact becomes inactive as it is replaced by its decomposed parts; in the third case, a modified artifact becomes inactive as it is superseded by a new version of itself.

The artifacts in the System Space are interconnected through traceability links. Traceability links in the System Space connect: (i) active artifacts and active artifacts, (ii) active artifacts and inactive artifacts, and (iii) inactive artifacts and inactive artifacts. There may be Permission links and Accountability links leaving or entering the System Space to/from the Actors Space.

6.5.1 Not Using the Homologation Process

The System Space is unchanged if a project chooses not to use the Homologation Process. Its division into Active System and Inactive System is still valid, since it is a concept modeled on existing practices; a development process has elements which are not used anymore (inactive) and elements which are part of the project "in development" (active). If a project discards the Homologation Process, it may activate artifacts directly, after an action takes place; for instance, after an artifact is created it is automatically added to the Active System. The same is true for the Removal Process; artifacts may be deactivated directly.

6.6 Traceability Space: Definition

Definition 6.6.1. A Traceability Space is a five-tuple $TS = (AS, SS, RS, PS, IEL)$, where:

- AS is an Actors Space;
- SS is a System Space;
- RS is a Rules Space;
- PS is a Processes Space;
- IEL is a set of Inter-Space Traceability Links.

Definition 6.6.2. A Traceability Space k (TS_k) represents a specific Traceability Space belonging to a project k; i.e., TS_k is an instance of a Traceability Space.

Definition 6.6.3. An Actors Space is a two-tuple $AS = (A, IAL)$, where: A is a set of Actors and IAL is a set of Intra-Space Traceability Links.[1]

The set of Actors $A = \{a_1, \ldots, a_n\}$, $n \geq 1$, is composed of all Actors interacting in the Traceability Space; i.e., Actors participating in the project being modeled by the Traceability Space. An Actor a_i, $i \leq n$, is a representation of an agent in the project. An Actor is modeled as a set $a = \{ia_1, \ldots, ia_m\}$, $m \geq 1$, where each ia_j, $j \leq m$, is a Block of Information containing identifiable information about the entity it represents. For instance, ia_1 may contain the name of the Actor, ia_2 may contain its employee number, etc. The concept of Blocks of Information will be discussed in more detail in the definition of the next Space.

The set of Intra-Space Traceability Links $IAL = \{ial_1, \ldots, ial_p\}$, $p \geq 1$, contains all links which interconnect the Actors in A. Each link ial_l, $l \leq p$, is a Traceability Link representing a relation between two distinct Actors in the Actors Space. The concept of Traceability Link will be discussed in more detail in the definition of the next Space.

[1] The Metamodel currently does not have intra-space traceability links for the Actors Space; these exist as a concept and may be added in the future.

Definition 6.6.4. A System Space is a three-tuple $SS = (AcS, InS, IESL)$, where: AcS is the Active System, InS is the Inactive System, and $IESL$ is a set of Inter-System Traceability Links.

The Active System is a two-tuple $AcS = (Art, IASL)$, where: Art is a set of Artifacts and $IASL$ is a set of Intra-System Traceability Links. The set of Artifacts $Art = \{\alpha_1, \ldots, \alpha_n\}$, $n \geq 1$, contains all active Artifacts in the project; i.e., Artifacts which are currently in use by the project. An Artifact α_i, $i \leq n$, is a representation of an product of the project. Each Artifact $\alpha = \{i_1, \ldots, i_m\}$, $m \geq 1$, is a set of Blocks of Information. A Block of Information is a sequence of arbitrary types, such as characters, forming an unit of relevant information. The granularity of a block of information may be highly variable, depending on necessity, context, domain, and so on; e.g., a Block of Information may be: a method in a C++ class, a C++ class, a .cpp file, part of a requirement document, etc. The set of Intra-System Traceability Links $IASL = \{iasl_1, \ldots, iasl_p\}$, $p \geq 1$, contains all links which interconnect the Artifacts in AcS. Each link $iasl_l$, $l \leq p$, is a Traceability Link representing a relation between two distinct Artifacts in the Active System. A Traceability Link is a three-tuple $tl = (Source, Target, Extra)$, where: $Source$ and $Target$ are the origin artifact and the destination artifact, respectively, of the Traceability Link. $Extra$ is a set of Blocks of Information containing extra information contained in the link; e.g., an Accountability link may have a Block of Information containing a date which indicates when the action took place.

The Inactive System is a two-tuple $InS = (Art, IASL)$, where: Art is a set of Artifacts and $IASL$ is a set of Intra-System Traceability Links. These are defined analogously to the previously defined sets having the same initialism: the set of Artifacts Art contains all inactive Artifacts in the project; the set of Intra-System Traceability Links $IASL$ contains all links which interconnect the Artifacts in InS.

The set of Inter-System Traceability Links $IESL = \{iesl_1, \ldots, iesl_q\}$, $q \geq 1$, contains all links which interconnect the Artifacts in the Active System and the Artifacts in the Inactive System; i.e., each Traceability Link $iesl_l$, $l \leq q$, is a Traceability Link representing a relation between two distinct Artifacts: one Artifact which belongs to the Active System and another Artifact which belongs to the Inactive System.

Definition 6.6.5. A Rules Space $RS = \{r_1, \ldots, r_n\}$, $n \geq 1$, contains a set of Rules for actions in the System Space. Each Rule is a 5-tuple $r = (Who, Actions, What, When)$, where: Who lists the Actors to whom the Rules applies. It may be a set of Actors, a set of roler, or a set of properties which Actors may have. The Rule is valid for all actors in the set or having the roles or properties described; e.g., Actor a, the set of Actors A, Actors who are "Developers", Actors who worked for more than 5 years in the company, Actors who held the position hp for the last 2 years, etc. Who can not be an empty set of elements, be Actors, roles, or properties.

$Actions$ defines the set of actions allowed by the Rule; e.g., the actions of decomposition and modification. $Actions$ can not be an empty set of elements.

$What$ defines the allowed target Artifacts for the elements of $Actions$. It may be a set of Artifacts, or a property which Artifacts may have; e.g., Artifact α, Artifacts having property pr, Artifacts not modified during the last year, Artifacts of type ty, etc. $What$ can not be an empty set of elements.

$When$ defines a date, or time window, in which the actions listed in *Actions* may occur; e.g., the actions may only happen during a certain time window t or at date d.

Definition 6.6.6. A Processes Space $PS = \{p_1, \ldots, p_n\}$, $n \geq 1$, contains a set of Processes done by Actors in the Actors Space to ensure Traceability Consistency and System Consistency (see Section 3.6 in Chapter 3). Processes and actions are integrated; i.e., Processes will not happen after an action is done. Each process combines an Action and the necessary instructions to ensure consistency given this action. Each Process p_i, $i \leq n$, is a two-tuple $p = (Act, Inst)$, where: *Act* is the action being performed in the Process and $Inst$ is the sequence of instructions to be done whenever the action defined by *Act* is performed.

Act defines the action which the Actor(s) in the Actors Space wishes to do; e.g., Modification or Creation.

$Inst$ determines the necessary steps to ensure consistency, given: (i) the action being performed, and (ii) the current state of the System Space; e.g., given the action of Removal, it is necessary to create a new Artifact, taking into account the current System Space, to avoid leaving two Artifacts in the Active System in the System Space with missing dependencies.

Definition 6.6.7. The Inter-Space Traceability Links is a two-tuple $IEL = (AL, PL)$, where: AL is a set of Accountability links and PL is a set of Permission links (see Section 3.3 in Chapter 3).

The set of Accountability links $AL = \{al_1, \ldots, al_n\}$, $n \geq 1$, contains all Accountability links which interconnect the Actors in the Actors Space and the Artifacts in the System Space. Each link al_i, $i \leq n$, is a Traceability Link representing a relation of Accountability between an Actor in the Actor Space and an Artifact for an Action performed in the System Space; e.g., Actor a performed the Action of modification on Artifact α.

The set of Permission links $PL = \{pl_1, \ldots, pl_m\}$, $m \geq 1$, contains all Permission links which interconnect the Actors in the Actors Space and the Artifacts in the System Space. Each link pl_i, $i \leq m$, is a Traceability Link representing a relation of Permission between an Actor in the Actor Space and an Artifact in the System Space; e.g., Actor a may perform the Action of modification on Artifact α.

6.7 Actions: Definitions

Actions are strongly related to other elements of the traceability model; the actions considered by a traceability model determine many of its link types and all of its processes. For instance, link types may trace accountability for actions, and a process is necessary for each action performed in the System Space (see Section 3.2.2 in Chapter 3).

Given the definition of the Traceability Space, each action contemplated by the Metamodel is defined in this section.

6.7.1 Modification

Definition 6.7.1. The Action of Modification is defined by the function *Modification*: $(TS_k, \{a_1, \ldots, a_n\} \in AS_k, \{i_1, \ldots, i_m\} \in \alpha \in SS_k, \{j_1, \ldots, j_p\}) \to TS'_k$, where: TS_k is the Traceability Space where the Action will be performed, $\{a_1, \ldots, a_n\}$ is the set of Actors who will perform the Action, $\{i_1, \ldots, i_m\}$ is the set of Blocks of Information to be modified, and $\{j_1, \ldots, j_p\}$ is the set of Blocks of Information which describes the modification to be performed on $\{i_1, \ldots, i_m\}$.

Let Mod: $(x, y) \to x'$ be a function which, given two sets of Blocks of Information as arguments, returns a modification of the first set according to instructions from the second set.

The function $Modification$ leads to changes in SS_k and IEL_k resulting in $TS'_k = (AS_k, SS'_k, RS_k, PS_k, IEL'_k)$, where: $InS'_k \in SS'_k = InS_k \cup \alpha$. Let $Mod(\alpha, \{j_1, \ldots, j_p\}) = \alpha'$; $AcS'_k \in SS'_k = (AcS_k \setminus \alpha) \cup \alpha'$. $IESL'_k \in SS'_k = IESL_k \cup \{new_iesl_1, new_iesl_2\}$, where $\{new_iesl_1, new_iesl_2\}$ are Evolution links between α and α'. $AL'_k \in IEL'_k = \{new_al_1, \ldots, new_al_q\} \cup AL_k$, where $\{new_al_1, \ldots, new_al_q\}$, $q = 2 \cdot n$, are Accountability links between $\{a_1, \ldots, a_n\}$ and α'.

6.7.2 Removal

Definition 6.7.2. The Action of Removal is defined by the function *Removal*: $(TS_k, \{a_1, \ldots, a_n\} \in AS_k, \alpha \in SS_k) \to TS'_k$. where: TS_k is the Traceability Space where the Action will be performed, $\{a_1, \ldots, a_n\}$ is the set of Actors who will perform the Action, and α is the Artifact to be removed.

The function *Removal* leads to changes in SS_k and IEL_k resulting in $TS'_k = (AS_k, SS'_k, RS_k, PS_k, IEL'_k)$, where: $InS'_k \in SS'_k = InS_k \cup \alpha$ and $AcS'_k \in SS'_k = AcS_k \setminus \alpha$. $AL'_k \in IEL'_k = \{new_al_1, \ldots, new_al_q\} \cup AL_k$, where $\{new_al_1, \ldots, new_al_q\}$, $q = 2 \cdot n$, are Accountability links between $\{a_1, \ldots, a_n\}$ and α.

6.7.3 Application

Definition 6.7.3. The Action of Application is defined by the function *Application*: $(TS_k, \{a_1, \ldots, a_n\} \in AS_k, \alpha \in SS_k) \to TS'_k$, where: TS_k is the Traceability Space where the Action will be performed, $\{a_1, \ldots, a_n\}$ is the set of Actors who will perform the Action, and α is the Artifact to be applied.

The Action of Application may result in the creation of one, or more, Artifacts.

If the Action of Application results in the creation of Artifacts, let $\{\beta_1, \ldots, \beta_m\}$, $m \geq 1$, be the set of these Artifacts. The function *Application* leads to changes in SS_k and IEL_k resulting in $TS'_k = (AS_k, SS'_k, RS_k, PS_k, IEL'_k)$, where: $AcS'_k \in SS'_k = AcS_k \cup \{\beta_1, \ldots, \beta_m\}$, $IESL'_k \in SS'_k = IESL_k \cup \{new_iesl_1, \ldots, new_iesl_p\}$, where $\{new_iesl_1, \ldots, new_iesl_p\}$, $p = 2 \cdot m$ are Action Outcome links between α and $\{\beta_1, \ldots, \beta_m\}$. $AL'_k \in IEL'_k = \{new_al_1, \ldots, new_al_q\} \cup AL_k$, where $\{new_al_1, \ldots, new_al_q\}$, $q = 2 \cdot n$, are Accountability links between $\{a_1, \ldots, a_n\}$ and α.

If the Action of Application does not result in the creation of Artifacts: the function *Application* leads to changes in IEL_k resulting in $TS'_k = (AS_k, SS_k, RS_k, PS_k, IEL'_k)$,

where: $AL'_k \in IEL'_k = \{new_al_1, \ldots, new_al_q\} \cup AL_k$, where $\{new_al_1, \ldots, new_al_q\}$, $q = 2 \cdot n$, are Accountability links between $\{a_1, \ldots, a_n\}$ and α.

6.7.4 Decomposition

Definition 6.7.4. The Action of Decomposition is defined by the function *Decomposition*: $(TS_k,\ \{a_1, \ldots, a_n\} \in AS_k,\ \alpha \in SS_k,\ \{j_1, \ldots, j_p\}) \to TS'_k$, where: TS_k is the Traceability Space where the Action will be performed, $\{a_1, \ldots, a_n\}$ is the set of Actors who will perform the Action, α is the Artifact to be decomposed, and $\{j_1, \ldots, j_p\}$ is the set of Blocks of Information which describes the decomposition.

Let *Dec*: $(\gamma, \{l_1, \ldots, l_p\}) \to \{\gamma_1, \ldots, \gamma_n\}$ be the function which, given two sets of Blocks of Information, returns a decomposition of the first set, according to instructions from the second set.

The function *Decomposition* leads to changes in SS_k and IEL_k resulting in $TS'_k = (AS_k, SS'_k, RS_k, PS_k, IEL'_k)$, where: $InS'_k \in SS'_k = InS_k \cup \alpha$ and $AcS'_k \in SS'_k = AcS_k \setminus \alpha$. Let $Dec(\alpha, \{l_1, \ldots, l_p\}) = \{\alpha_1, \ldots, \alpha_m\}$, $m \geq 2$; $AcS'_k \in SS'_k = AcS_k \cup \{\alpha_1, \ldots, \alpha_m\}$. $IESL'_k \in SS'_k = IESL_k \cup \{new_iesl_1, \ldots, new_iesl_r\}$, $r = 2 \cdot m$, where new_iesl_i and $new_iesl_{(r/2)+i}$, $i \leq r$, are Composition links between α and α_i. $AL'_k \in IEL'_k = \{new_al_1, \ldots, new_al_q\} \cup AL_k$, where $\{new_al_1, \ldots, new_al_q\}$, $q = 2 \cdot n$, are Accountability links between $\{a_1, \ldots, a_n\}$ and α.

6.7.5 Reification

Definition 6.7.5. The Action of Reification is defined by the function $Reification$: $(TS_k,\ \{a_1, \ldots, a_n\} \in AS_k,\ \{j_1, \ldots, j_p\}) \to TS'_k$, where: TS_k is the Traceability Space where the Action will be performed, $\{a_1, \ldots, a_n\}$ is the set of Actors who will perform the Action, $\{j_1, \ldots, j_p\}$ is the set of Blocks of Information which will be used to create a new Artifact. The set $\{j_1, \ldots, j_p\}$ is composed of Blocks of Information from one, or more, Artifacts.

Let $Reif$: $(\{j_1, \ldots, j_p\}) \to \alpha$ be a function in which, given a set of Blocks of Information, returns a new Artifact composed of reifications of all Blocks of Information given as argument. A resulting reified Block of Information may be created by using one, or more, Blocks of Information.

The function $Reification$ leads to changes in SS_k resulting in $TS'_k = (AS_k,\ SS'_k,\ RS_k,\ PS_k,\ IEL_k)$, where: let $Reif(\{j_1, \ldots, j_p\}) = \alpha$; $AcS'_k \in SS'_k = AcS_k \cup \alpha$. Let $\beta = \{\beta_1, \ldots, \beta_m\}$ be the set of Artifacts where $\forall \beta_i \in \beta,\ \exists j_l \in \beta_i,\ l \leq p$; $IESL'_k \in SS'_k = IESL_k \cup \{new_iesl_1, \ldots, new_iesl_r\}$, $r = 2 \cdot m$, where new_iesl_i and $new_iesl_{(r/2)+i}$, $i \leq r$, are Evolution links between β_i and α.

6.7.6 Creation

Definition 6.7.6. The Action of Creation is defined by the function *Creation*: $(TS_k,\ \{a_1, \ldots, a_n\} \in AS_k) \to TS'_k$, where: TS_k is the Traceability Space where the Action will be performed and $\{a_1, \ldots, a_n\}$ is the set of Actors who will perform the Action.

Let *Cre*: $a \to x$ be a function which, given a set of Actors as argument, returns a set of Blocks of Information; i.e., it returns a new Artifact created by the Actors.

The function $Creation$ leads to changes in SS_k and IEL_k resulting in $TS'_k = (AS_k, SS'_k, RS_k, PS_k, IEL'_k)$, where: let $Cre(\{a_1, \ldots, a_n\}) = \alpha$; $AcS'_k \in SS'_k = AcS_k \cup \alpha$, and $AL'_k \in IEL'_k = \{new_al_1, \ldots, new_al_q\} \cup AL_k$, where $\{new_al_1, \ldots, new_al_q\}$, $q = 2 \cdot n$, are Accountability links between $\{a_1, \ldots, a_n\}$ and α.

6.7.7 Homologation

Definition 6.7.7. The Action of Homologation is defined by the function $Homologation$: $(TS_k, \{a_1, \ldots, a_n\} \in AS_k, \alpha \in SS_k) \to TS'_k$, where: TS_k is the Traceability Space where the Action will be performed, $\{a_1, \ldots, a_n\}$ is the set of Actors who will perform the Action and α is the Artifact to be homologated.

The function $Homologation$ is not used directly by Actors, instead it is used by other functions to homologate Artifacts; Artifacts which are homologated are added to the Active System of a Traceability Space. For instance, the $Modification$ function uses the $Homologation$ function to activate α', the modified version of α.

If $\{a_1, \ldots, a_n\}$ approves the homologation of α, the function $Homologation$ leads to changes in SS_k and IEL_k resulting in $TS'_k = (AS_k, SS'_k, RS_k, PS_k, IEL'_k)$, where: $InS'_k \in SS'_k = InS_k \setminus \alpha$, $AcS'_k \in SS'_k = AcS_k \cup \alpha$, and $AL'_k \in IEL'_k = \{new_al_1, \ldots, new_al_q\} \cup AL_k$, where $\{new_al_1, \ldots, new_al_q\}$, $q = 2 \cdot n$, are Accountability links between $\{a_1, \ldots, a_n\}$ and α.

6.7.8 Activation

Definition 6.7.8. The Action of Activation is defined by the function $Activation$: $(TS_k, \{a_1, \ldots, a_n\} \in AS_k, \alpha \in SS_k, \{j_1, \ldots, j_p\}) \to TS'_k$, where: TS_k is the Traceability Space where the Action will be performed, $\{a_1, \ldots, a_n\}$ is the set of Actors who will perform the Action, α is the Artifact to be Activated, and $\{j_1, \ldots, j_p\}$ is an optional set of Blocks of Information which describes the modification to be performed on α.

The Action of Activation may be contingent upon a modification of the target Artifact; if $\{j_1, \ldots, j_p\} = \emptyset$, α will be activated without changes.

The function $Activation$ is used by Actors to add inactive Artifacts to the Active System; these are new Artifacts or Artifacts which were previously rejected by the function of $Homologation$.

If $\{j_1, \ldots, j_p\} = \emptyset$, the function $Activation$ leads to changes in SS_k resulting in $TS'_k = (AS_k, SS'_k, RS_k, PS_k, IEL_k)$, where: $InS'_k \in SS'_k = InS_k \setminus \alpha$ and $AcS'_k \in SS'_k = AcS_k \cup \alpha$. The Accountability for this Action is provided by the $Homologation$ function, thus there are no changes to IEL_k for this function.

If $\{j_1, \ldots, j_p\} \neq \emptyset$, the function $Activation$ leads to changes in SS_k resulting in $TS'_k = (AS_k, SS'_k, RS_k, PS_k, IEL_k)$, where: let α' be the result of the modification of α[2]; $AcS'_k \in SS'_k = AcS_k \cup \alpha'$. The Artifact α stays in the Inactive System.

[2]The Artifact α' is created as a result of the $Modification$ function.

6.8 The Traceability Space and Metrics

Relevant information may be inferred by analyzing the Traceability Space of a project in development; for instance: an Actor having a significant number of Accountability links to Artifacts in the Active System, when compared to other Actors, may be overworked; an Actor having a significant number of Accountability links to Artifacts in the Inactive System is someone who contributed substantially to the project in the past; an Artifact having a significant number of Constraint links of the dependency type is highly relevant to a project and its removal may be costly; and the decision to modify an Artifact may be influenced by how highly connected it is to other Artifacts in the Active System; a high number of rejected Artifacts in the Inactive System, belonging to the same Actor, may be investigated to evaluate the necessity of providing training for this Actor.

Chapter 7

Artifact Types

Artifacts are the products generated during the software development process; there may be different activities (or phases) of development, given the development model used. Thus, Artifacts may be classified according to the development activity they belong to. For instance, an Artifact modeling a requirement and an Artifact modeling a test case belong to the requirements engineering activity and the Verification & Validation and Testing activity, respectively.

Our Reference Model described in Chapter 3 suggests five basic Artifact types for traceability: Pre-Requirements, Requirements, Design, Implementation, and Verification & Validation and Testing; these cover the most common activities in software development. Our Metamodel uses the same Artifact types and adds a sixth type: the Rationale Artifact.

The Rationale Artifact provides rationale and/or description concerning an action; the rationale is a justification (why) for an Action, and the description contains instructions to perform an Action (what and how). Therefore, it enables the possibility of saving the reasons for actions performed in the System Space, and details about how the actions where done.

Since the Rationale Types justify and/or describe actions, it is necessary to have a Rationale type for each Action being traced. Seven Rationale Types are defined in this chapter.[1]

This chapter is structured as follows: Section 7.1 lists the non-rationale artifact types of the Metamodel, and Section 7.2 defines the rationale artifact types of the Metamodel.

7.1 Non-Rationale Artifact Types

The Non-Rationale Artifact Types of the Metamodel are: Pre-Requirements Engineering Artifact, Requirements Engineering Artifact, Design Artifact, Implementation Artifact, and Verification & Validation and Testing Artifact. These are described in Chapter 3.

[1] The Traceability Space of the Metamodel considers eight actions; however, the same Rationale type is used for the Action of creation and the Action of reification.

7.2 Rationale Artifact Types

The Rationale Artifact types of the Metamodel are: Rationale for Modification (RM), Rationale for Removal (RR), Rationale for Decomposition (RD), Rationale for Homologation or Rejection (RHR), Rationale for Creation (RC), Rationale for Application (RAp), and Rationale for Activation (RAc). The Rationale for Creation and the Rationale for Application are optional types.

7.2.1 Rationale for Modification

A Rationale for Modification provides rationale for the modification of an Artifact and describes the modifications to be made. The description contain what changes are to be done in the Artifact and how they should be done.

The Rationale for Modification is defined as: $RM = R \cup (DW \cup DH)$, where R, DW, and DH are sets of Blocks of Information. The set R is the justification for the modification. The sets DW e DH are the description of the modification, where: DW defines what changes should be done and DH defines how the changes should be done. The set DH is optional.

The Rationale for Modification becomes part of the Active System when an Action of modification is being done; when the Action is finalized, the Rationale for Modification is moved to the Inactive System.

7.2.2 Rationale for Removal

A Rationale for Removal provides rationale for the removal of an Artifact and may describe how the removal should be done; the removal may require changes to other Artifacts, thus, a Rational for Removal may describe which Artifacts should be modified, what should be modified, and how.

The description is optional; it may be necessary if the removal requires changes to related Artifacts. For instance, the removal of a function in an Implementation Artifact may require removing calls in other Artifacts. Consequently, the Rationale for Removal has two possible formats: $RR = R$, or $RR = R \cup (DW \cup DH)$, where R, DW, and DH are sets of Blocks of Information. The set R is the justification for the removal. The sets DW e DH are the description of the removal, where: DW defines which Artifacts should be modified and what changes should be done, and DH defines how these changes should be done. In the second format, the set DH is optional.

The Rationale for Removal becomes part of the Active System when an Action of removal is being done; when the Action is finalized, the Rationale for Removal is moved to the Inactive System.

7.2.3 Rationale for Decomposition

A Rationale for Decomposition provides rationale for the decomposition of an Artifact and describes how the decomposition should be made. The description contains which information should go into each new Artifact. For instance, let α be a Rationale for

Decomposition and let $\beta = \{i_1, \ldots, i_n\}$ be an Artifact to be decomposed; the description in α could define the creation of two artifacts β_1 and β_2, where: $\beta_1 = \{i_1, \ldots, i_p\}$, $p < n$, and $\beta_2 = \{i_{p+1}, \ldots, i_n\}$.

The Rationale for Decomposition is defined as: $RD = R \cup D$, where R and D are sets of Blocks of Information. The set R is the justification for the decomposition. The set D defines how many Artifacts should be created and which Blocks of Information from the decomposed Artifact should go to each new Artifact.

The Rationale for Decomposition becomes part of the Active System when an Action of decomposition is being done; when the Action is finalized, the Rationale for Decomposition is moved to the Inactive System.

7.2.4 Rationale for Homologation or Rejection

A Rationale for Homologation or Rejection provides rationale for the Homologation or Rejection of an Artifact; i.e., an Artifact which went through the Action of Homologation. The Rationale for Homologation or Rejection explains why an Artifact was homologated, and added to the Active System, or why an Artifact was rejected, and stayed in the Inactive System.

The Rationale for Homologation or Rejection is defined as: $RHR = R$, where R is a set of Blocks of Information. The set R is the justification for the homologation or the rejection of the Artifact.

The Rationale for Homologation or Rejection never becomes part of the Active System; it is kept in the Inactive System. It can be conferred through its relation to the homologated, or rejected, Artifact.

7.2.5 Rationale for Creation

A Rationale for Creation is an optional Rationale type which provides rationale for the creation of an Artifact and may describe the Artifact to be created. This Rationale Type is created to justify the creation of an Artifact, or to justify the creation of an Artifact and describe how this new Artifact should be; i.e., detail the content of the desired Artifact.

The description is optional; a Rationale for Creation may provide reasons for the creation of an Artifact without providing instructions for its creation. Consequently, the Rationale for Creation has two possible formats: $RC = R$, or $RC = R \cup D$, where R and D are sets of Blocks of Information. The set R is the justification for creating the Artifact and the set D is the description of the Artifact to be created.

This Artifact type is optional; it is provided to enable users to justify, or justify and describe, the creation of Artifacts. It is not used in the processes in Chapter 9.

A Rationale for Creation may also be used to provide justifications concerning the creation of a reified Artifact.

7.2.6 Rationale for Application

A Rationale for Application is an optional Rationale Type which provides rationale for the application of an Artifact.

The Rationale for Application is defined as: $RAp = R$, where R is a set of Blocks of Information. The set R is the justification for the application.

This Artifact type is optional; it is not used in the processes in Chapter 9.

7.2.7 Rationale for Activation

A Rationale for Activation provides rationale for the activation of an Artifact and may also describe modifications to be done to the Artifact; i.e., the activation of an Artifact may be contingent on a modification which makes it useful for the Active System.

The description is optional; an Artifact can be activated without being modified. Consequently, the Rationale for Activation has two possible formats: $RAc = R$, or $RAc = R \cup (DW \cup DH)$, where R, DW, and DH are sets of Blocks of Information. The set R is the justification for the activation. The sets DW e DH are the description of the modification, where: DW defines what changes should be done and DH defines how the changes should be done. The set DH is optional.

The Rationale for Activation becomes part of the Active System when an Action of activation is being done; when the Action is finalized, the Rationale for Activation is moved to the Inactive System.

Chapter 8

Relations Modeled as Link Types

Link types are one of the main elements of traceability; they model the relations between artifacts and actors in a project. Each modeled relation has particular semantics concerning the elements it connects, and link types must be described taking into account these semantics.

Our Reference Model described in Chapter 3 suggests seven basic link types for traceability; these are generic link types having the goal of covering common relations between elements in a software development project. The link types of the Reference Model are: Evolution, Constraint, Accountability, Permission, Characterize Action, Action Outcome, and Composition. Since they are generic link types, most may be broken into more specific link types.

The Accountability link type should be divided into at least six types, one for each basic action (these are also discussed in the Reference Model); having link types for each basic action ensures accountability for actors every time an action is performed. The Evolution link type should be divided into at least two types; one link type to model propagation and change of information and one link type to model propagation and reification of information. The Constraint link type should be divided into at least two types: a link type modeling dependency between artifacts and another link type modeling conflict between artifacts. The Permission link type should be divided into at least six types, one for each basic action; this enables the ability to regulate, and trace, authorization to perform each basic action. The Characterize link type should be divided into at least six types, one for each basic action; having link types for each basic action ensures the possibility to trace rationale for each action performed. The Composition link type should be divided into at least two types; one link type to model the decomposition relation and one link type to model the "part of" relation between artifacts.

Each link type is semantically described and formalized to avoid ambiguities and enable its mapping to real relations.

A total of 59 link types are defined, satisfying the roles of each of the seven link types of the Reference Model; there are 30 distinct relations being represented by these link types. Table 8.1 shows the complete list of link types organized by its correspondent link type in the Reference Model.

Table 8.1: Link Types of the Reference Model × Metamodel Link types.

Reference Model	Metamodel
Evolution	ReifiedFrom, ReifiedTo, ModifiedFrom, and ModifiedTo.
Constraint	DependsOn, NecessaryFor, DependsOn^{Partial}, NecessaryFor^{Partial}, and ConflictsWith.
Accountability	HomologatedBy, Homologated, RejectedBy, Rejected, CreatedBy, Created, ModifiedBy, Modified, AppliedBy, Applied, DecomposedBy, Decomposed, RemovedBy, and Removed.
Permission	MayHomologate, MayBeHomologatedBy, MayModify, MayBeModifiedBy, MayRemove, MayBeRemovedBy, MayDecompose, MayBeDecomposedBy, MayApply, MayBeAppliedBy, MayActivate, and MayBeActivatedBy.
Characterize Action	DefinesModificationOf, ModificationDefinedBy, DefinesDecompositionOf, DecompositionDefinedBy, DefinesRemovalOf, RemovalDefinedBy, DefinesActivationOf, ActivationDefinedBy, DefinesCreationOf, CreationDefinedBy, SubjectToApplicationOf, AppliesTo, JustifiesHomologationOf, HomologationJustifiedBy, JustifiesRejectionOf, RejectionJustifiedBy, JustifiesApplicationOf, and ApplicationJustifiedBy.
Action Outcome	ApplicationProduced, and ProducedByApplicationOf.
Composition	DecomposedFrom, DecomposedTo, PartOf, and ComprisedOf.

This chapter is structured as follows: Section 8.1 discusses two possible ways to model relations between artifacts, Section 8.2 defines each link type of the Metamodel, and Section 8.3 provides closing remarks on the subject.

8.1 Relations Modeled as One or Two Link Types?

A relation may be interpreted as a single link type or as two link types, one in each opposite direction; however, a directed edge should be used to represent the relation. For instance, if a relation of dependency is modeled as an undirected edge, connecting both artifacts, it is not expressing what depends on what. If a directed edge is used, it is possible to know which element depends on another and which element is needed to fulfill this dependency.

Both representations – single link type and two link type – are equally valid to model relations between elements and are able to express the same quantity of traceability information. It may seem that using two link types is a richer way to represent relations but this is not true. To verify that, it is enough to read a single link type representing a relation as a sentence; for instance, “artifact α depends on artifact β”. From this sentence we are able to gather the information that α depends on β, and β is dependent on α; this is the same information recorded by a model using two link types.

On the other hand, (i) it may be counterintuitive to interpret a directed edge in its opposite direction, and (ii) it may give more flexibility to have two link types modeling a relation when constructing specific representations in a traceability model. To exemplify the latter, suppose that a traceability model creator wants to discard traceability information which connects Active System artifacts to Inactive System artifacts, but only in this direction; i.e., information going from the Active System to the Inactive System. Having two link types enables this possibility, since an inactive artifact is able to reference an active artifact without the opposite being true. Graphically, there is an edge leaving the Inactive System to the Active system but there is not an edge in the opposite direction.

This Metamodel models relations as two link types, one in each direction.

8.2 Link Types

Descriptions of the proposed link types, fulfilling the basic link types of the Reference Model, plus necessary definitions, are shown in this section. The link types are organized by link type of the Reference Model; i.e., each subsection is a link type of the Reference Model and contains the link types of the Metamodel which realize it.

Two starting definitions are necessary, given that each relation is modeled by two link types: *opposite link types* and *opposite links*.

Definition 8.2.1. Two link types are said to be *opposite link types* if they convey the same information in opposite directions.

For instance, let A_1 and A_2 be two artifact types and let L_1 be a link type starting in A_1 and having A_2 as its destination. Let L_1 convey the information that an instance of A_1 may depend on an instance of A_2; a link type L_2 starting in A_2 and having A_1 as its destination conveying the information that an instance of A_2 may be necessary for an instance of A_1 is an *opposite link type* to L_1.

Definition 8.2.2. Two links are said to be *opposite links* if they are instances of *opposite link types* and connect the same pair of artifacts.

For instance, let α and β be two artifacts and let e be a link starting in α and having β as its destination. Let e conveys the information that α depends on β; a link f starting in β and having α as its destination conveying the information that β is necessary for α is an *opposite link* to e.

The link types of the Metamodel are shown next. Definitions are provided as needed.

8.2.1 Evolution Link Types

The following link types belong to the Evolution link type of the Reference Model: ReifiedFrom, ReifiedTo, ModifiedFrom, and ModifiedTo.

Table 8.2 matches each link type to its opposite link type; the column assignment is not relevant, i.e., both link types are opposite to each other.

Table 8.2: Evolution Link Types.

Link Type	Opposite Link Type
ReifiedFrom	ReifiedTo
ModifiedFrom	ModifiedTo

Definition 8.2.3. An artifact α was reified from an artifact β if part of, or all of, α was constructed in a structured way from all of, or part of, β. Implicit information from β may be made explicit in α. Conversely, β was reified into α.

ReifiedFrom Link Type

Let α and β be two artifacts and let e be a link starting in α and having β as its destination. The link e is a *ReifiedFrom* link type if it conveys the information that α was reified from β.

For instance, let α_1 and α_2 be Requirements Engineering Artifacts and let β be a Design Artifact. Let all of α_1 and part of α_2 be used in the construction of β, i.e., information written in natural language from α_1 and α_2 was transformed into information written in a design representation language in β. Hence, there are two ReifiedFrom links which start in β and have α_1 and α_2 as their destination.

Formalization. Given Definition 8.2.3, let $Reif$: $\gamma \to \gamma'$ be the function which, given a set of blocks of information, returns a reification of this set. Let $\alpha = \{i'_1, \ldots, i'_n\}$, $n \geq 1$, and $\beta = \{i_1, \ldots, i_m\}$, $m \geq 1$, be artifacts where $\alpha \neq \beta$. Let SS_k be a System Space k, AS_k be the Active System of SS_k, and IS_k be the Inactive System of SS_k. If $\alpha \in AS_k$, $\alpha \subseteq \{i'_p, \ldots, i'_q\}$, $1 \leq p \leq q \leq n$, then $\exists \beta \in AS_k$, $\beta \subseteq \{i_r, \ldots, i_s\}$, $1 \leq r \leq s \leq m$: $Reif(i_r, \ldots, i_s) = (i'_p, \ldots, i'_q)$. If $\alpha \in IS_k$, $\alpha \subseteq \{i'_p, \ldots, i'_q\}$, $1 \leq p \leq q \leq n$, then $\exists \beta \in IS_k$, $\beta \subseteq \{i_r, \ldots, i_s\}$, $1 \leq r \leq s \leq m$: $Reif(i_r, \ldots, i_s) = (i'_p, \ldots, i'_q)$.

ReifiedTo Link Type

Let α and β be two artifacts and let e be a link starting in α and having β as its destination. The link e is a *ReifiedTo* link type if it conveys the information that α was reified into β.

The link type ReifiedTo is an opposite link type to the ReifiedFrom link type.

Formalization. Let $\alpha = \{i_1, \ldots, i_n\}$, $n \geq 1$, and $\beta = \{i'_1, \ldots, i'_m\}$, $m \geq 1$, be artifacts where $\alpha \neq \beta$. Let SS_k be a System Space k, AS_k be the Active System of SS_k, and IS_k be the Inactive System of SS_k. If $\alpha \in AS_k$, $\alpha \subseteq \{i_p, \ldots, i_q\}$, $1 \leq p \leq q \leq n$, then $\exists \beta \in AS_k$, $\beta \subseteq \{i'_r, \ldots, i'_s\}$, $1 \leq r \leq s \leq m$: $Reif(i_p, \ldots, i_q) = (i'_r, \ldots, i'_s)$. If $\alpha \in IS_k$, $\alpha \subseteq \{i_p, \ldots, i_q\}$, $1 \leq p \leq q \leq n$, then $\exists \beta \in IS_k$, $\beta \subseteq \{i'_r, \ldots, i'_s\}$, $1 \leq r \leq s \leq m$: $Reif(i_p, \ldots, i_q) = (i'_r, \ldots, i'_s)$.

Definition 8.2.4. An artifact α was modified from an artifact β if α was created by copying β and subsequently modificating this copy according to previously defined instructions; i.e., α is a new, modified, version of the artifact β. Conversely, it is said β was

modified into α. The modification of an artifact may change existing information and/or add new information.

ModifiedFrom Link Type

Let α and β be two artifacts and let e be a link starting in α and having β as its destination. The link e is a *ModifiedFrom* link type if it conveys the information that α was modified from β; α is the new version of β.

Formalization. Given Definition 8.2.4, let Mod: $(x, y) \to x'$ be the function which, given two sets of blocks of information as arguments, returns a modification of the first set according to instructions from the second set. Let $\alpha = \{i_1, \ldots, i_p\} \cup \{i'_q, \ldots, i'_n\}$, $1 < p < q \leq n$, and $\beta = \{j_1, \ldots, j_m\}$, $m \geq 1$, be artifacts, where $\alpha \neq \beta$ and $\{i_1, \ldots, i_p\}$ may be empty. Let $\{l_1, \ldots, l_p\}$, $p \geq 1$, be a set of blocks of information describing a modification. Let SS_k be a System Space k. If $\alpha \in SS_k$, then $((\exists \beta \in SS_k : \beta \subseteq \{j_r, \ldots, j_s\}, 1 \leq r \leq s \leq m) \wedge (\exists \{l_1, \ldots, l_p\} \in SS_k)) : Mod(\{j_r, \ldots, j_s\}, \{l_1, \ldots, l_p\}) = \{i'_q, \ldots, i'_n\}$.

The set of blocks $\{i_1, \ldots, i_p\}$ represents unadultered information from β in α; therefore, if $\{i_1, \ldots, i_p\} = \emptyset \implies \{j_r, \ldots, j_s\} = \{j_1, \ldots, j_m\}$.

ModifiedTo Link Type

Let α and β be two artifacts and let e be a link starting in α and having β as its destination. The link e is a *ModifiedTo* link type if it conveys the information that α was modified into β; α is the previous version of β.

The link type ModifiedTo is an opposite link type to the ModifiedFrom link type.

Formalization. Let $\alpha = \{i_1, \ldots, i_n\}$, $n \geq 1$ and $\beta = \{j_1, \ldots, j_p\} \cup \{j'_q, \ldots, j'_m\}$, $1 < p < q \leq m$ be artifacts, where $\alpha \neq \beta$ and $\{j_1, \ldots, j_p\}$ may be empty. Let $\{l_1, \ldots, l_p\}$, $p \geq 1$, be a set of blocks of information describing a modification. Let SS_k be a System Space k. If $\alpha \in SS_k$, $\alpha \subseteq \{i_r, \ldots, i_s\}$, $1 \leq r \leq s \leq n$, then $((\exists \beta \in SS_k) \wedge (\exists \{l_1, \ldots, l_p\} \in SS_k))$ $: Mod(\{i_r, \ldots, i_s\}, \{l_1, \ldots, l_p\}) = \{j'_q, \ldots, j'_m\}$.

These four link types fulfill the Evolution link type of the Reference Model by conveying that information was reified, or modified, and propagated from one artifact to another.

8.2.2 Constraint Link Types

The following link types belong to the Constraint link type of the Reference Model: DependsOn, NecessaryFor, DependsOn^{Partial}, NecessaryFor^{Partial}, and ConflictsWith.

Table 8.3 matches each link type to its opposite link type; the column assignment is not relevant, i.e., both link types are opposite to each other.

Table 8.3: Constraint Link Types.

Link Type	Opposite Link Type
DependsOn	NecessaryFor
DependsOn^{Partial}	NecessaryFor^{Partial}
ConflictsWith	ConflictsWith

Definition 8.2.5. An artifact α is satisfied by an artifact β if a condition, or a set of conditions, required by α is fulfilled by β.

DependsOn Link Type

Let α and β be two artifacts and let e be a link starting in α and having β as its destination. The link e is a *DependsOn* link type if it conveys the information that α is satisfied by β.

Dependency relations may appear in many ways, such as: an artifact may depend on characteristics of another artifact to run correctly (existential dependency); the application of an artifact may depend on the previous application of other artifacts (depends on previous actions); among other kinds of dependencies.

Formalization. Given Definition 8.2.5, let Sat: $(x, y) \rightarrow \{True, False\}$ be a predicate, which given two sets of blocks of information as arguments, returns True if the second set satisfies the first set and returns False otherwise. Let $\alpha = \{i_1, \ldots, i_n\}$, $n \geq 1$, and $\beta = \{j_1, \ldots, j_m\}$, $m \geq 1$, be artifacts where $\alpha \neq \beta$. Let SS_k be a System Space k. If $\alpha \in SS_k$, $\alpha \subseteq \{i_p, \ldots, i_q\}$, $1 \leq p \leq q \leq n$, then $\exists \beta \in SS_k$, $\beta \subseteq \{j_r, \ldots, j_s\}$, $1 \leq r \leq s \leq m$: $Sat(\{i_p, \ldots, i_q\}, \{j_r, \ldots, j_s\}) = True$.

NecessaryFor Link Type

Let α and β be two artifacts and let e be a link starting in α and having β as its destination. The link e is a *NecessaryFor* link type if it conveys the information that α satisfies β.

The link type NecessaryFor is an opposite link type to the to the DependsOn link type.

Formalization. Let $\alpha = \{i_1, \ldots, i_n\}$, $n \geq 1$, and $\beta = \{j_1, \ldots, j_m\}$, $m \geq 1$, be artifacts where $\alpha \neq \beta$. Let SS_k be a System Space k. If $\alpha \in SS_k$, $\alpha \subseteq \{i_p, \ldots, i_q\}$, $1 \leq p \leq q \leq n$, then $\exists \beta \in SS_k$, $\beta \subseteq \{j_r, \ldots, j_s\}$, $1 \leq r \leq s \leq m$: $Sat(\{j_r, \ldots, j_s\}, \{i_p, \ldots, i_q\}) = True$.

DependsOn^{Partial} Link Type

Let α and β be two artifacts and let e be a link starting in α and having β as its destination. The link e is a *DependsOn*^{Partial} link type if it conveys the information that α is satisfied by β, but not only by it.

This link type is a variation of the DependsOn link type. Every artifact identified by DependsOn^{Partial} links is able to, independently, satisfy another artifact. For instance,

suppose an artifact α has two DependsOn^{Partial} links having artifacts β_1 and β_2 as their destination; the removal of β_1 does not cause issues for α, since it is also satisfied by β_2.

A DependsOn^{Partial} link exists if there are at least two artifacts satisfying the same artifact by using the same information; i.e., the existence of this link type implies that there is redundant information in the System Space.

Formalization. Let $\alpha = \{i_1, \ldots, i_n\}$, $n \geq 1$, $\beta = \{j_1, \ldots, j_m\}$, $m \geq 1$, and $\gamma = \{l_1, \ldots, l_w\}$, $w \geq 1$, be artifacts where $\alpha \neq \beta$, $\alpha \neq \gamma$, and $\beta \neq \gamma$. Let SS_k be a System Space k. If $\alpha \in SS_k$, $\alpha \subseteq \{i_p, \ldots, i_q\}$, $1 \leq p \leq q \leq n$, then $(\exists \beta \in SS_k,\ \beta \subseteq \{j_r, \ldots, j_s\},\ 1 \leq r \leq s \leq m)$ $\wedge$ $(\exists \gamma \in SS_k,\ \gamma \subseteq \{j_r, \ldots, j_s\},\ 1 \leq r \leq s \leq w) : Sat(\{i_p, \ldots, i_q\}, \{j_r, \ldots, j_s\}) = True$.

NecessaryFor^{Partial} Link Type

Let α and β be two artifacts and let e be a link starting in α and having β as its destination. The link e is a *NecessaryFor^{Partial}* link type if it conveys the information that α satisfies β, but α is not the only artifact which satisfies β.

The link type NecessaryFor^{Partial} is an opposite link type to the to the DependsOn^{Partial} link type.

Formalization. Let $\alpha = \{i_1, \ldots, i_n\}$, $n \geq 1$, $\beta = \{j_1, \ldots, j_m\}$, $m \geq 1$, and $\gamma = \{l_1, \ldots, l_w\}$, $w \geq 1$, be artifacts where $\alpha \neq \beta$, $\alpha \neq \gamma$, and $\beta \neq \gamma$. Let SS_k be a System Space k. If $\alpha \in SS_k$, $\alpha \subseteq \{i_p, \ldots, i_q\}$, $1 \leq p \leq q \leq n$, then $(\exists \beta \in SS_k,\ \beta \subseteq \{j_r, \ldots, j_s\},\ 1 \leq r \leq s \leq m)$ $\wedge$ $(\exists \gamma \in SS_k,\ \gamma \subseteq \{i_p, \ldots, i_q\},\ 1 \leq r \leq s \leq w) : Sat(\{j_r, \ldots, j_s\}, \{i_p, \ldots, i_q\}) = True$.

Definition 8.2.6. Two artifacts are in conflict with one another if they should not coexist in the Active System.

ConflictsWith Link Type

Let α and β be two artifacts and let e be a link starting in α and having β as its destination. The link e is a *ConflictsWith* link type if it conveys the information that α is in conflict with β.

There are two types of conflicts: contradictory information and redundant information; not all redundant information should be classified as conflict. Each type may be specified in the Blocks of Information, set apart for extra content, in traceability links.

The ConflictsWith link type is opposite to itself; hence, it does not have a distinct opposite link type. The Metamodel represents this relation as two ConflictsWith link types in opposite directions.

Formalization. Given Definition 8.2.6, let $Cflt$: $(x, y) \rightarrow \{True, False\}$ be a predicate, which given two sets of blocks of information as arguments, returns True if they conflict with each other and returns False otherwise. Let $\alpha = \{i_1, \ldots, i_n\}$, $n \geq 1$, and $\beta = \{j_1, \ldots, j_m\}$, $m \geq 1$, be artifacts where $\alpha \neq \beta$. Let SS_k be a System Space k, AS_k be the Active System of SS_k, and IS_k be the Inactive System of SS_k. If $\alpha \in AS_k$, $\alpha \subseteq \{i_p, \ldots, i_q\}$, $1 \leq p \leq q \leq n$, then $\exists \beta \in IS_k$, $\beta \subseteq \{j_r, \ldots, j_s\}$: $Cflt(\{i_p, \ldots, i_q\}, \{j_r, \ldots, j_s\}) = True$. If $\alpha \in IS_k$, $\alpha \subseteq \{i_p, \ldots, i_q\}$, $1 \leq p \leq q \leq n$,

then $\exists \beta \in SS_k$, $\beta \subseteq \{j_r, \ldots, j_s\} : Cflt(\{i_p, \ldots, i_q\}, \{j_r, \ldots, j_s\}) = True$.

These five link types fulfill the two roles of the Constraint link type of the Reference Model: the dependency constraint and the conflict constraint.

8.2.2.1 Multiple Dependencies to the Same Artifact

There may be artifacts which depend on multiple distinct information contained in a single artifact. For instance, an artifact α depends on two sets of distinct information in an artifact β. This situation may be modeled by: (i) using composition links and multiple constraint links for each identified part; or (ii) by using the Blocks of Information – set apart for extra content – in traceability links to identify which sets of information satisfy the artifact.

8.2.2.2 Transformation Between Non-Partial and Partial Links

DependsOn and NecessaryFor links become their partial versions if redundant – and necessary for an artifact – information is added to the System Space; conversely, DependsOn^{Partial} and NecessaryFor^{Partial} link types become their non-partial versions if they are the only links left indicating a specific necessary information in the System Space.

To exemplify, let α, β_1 and β_2 be artifacts in the Active System. Artifact α depends on information contained in β_1; thus, there is a NecessaryFor/DependsOn pair of links between α and β_1. If β_2 is modified to contain the same information contained in β_1 which is necessary for α, a DependsOn^{Partial}/NecessaryFor^{Partial} pair of links should be created between α and β_2, and the NecessaryFor/DependsOn pair of links between α and β_1 becomes a DependsOn^{Partial}/NecessaryFor^{Partial} pair of links.

If β_1 is modified to remove the information necessary for α, the DependsOn^{Partial}/NecessaryFor^{Partial} pair of links between α and β_1 is deleted and the DependsOn^{Partial}/NecessaryFor^{Partial} pair of links between α and β_2 becomes a NecessaryFor/DependsOn pair of links.

8.2.3 Accountability Link Types

The following link types belong to the Accountability link type of the Reference Model: CreatedBy, Created, ModifiedBy, Modified, RemovedBy, Removed, AppliedBy, Applied, DecomposedBy, Decomposed, HomologatedBy, Homologated, RejectedBy, and Rejected.

Table 8.4 matches each link type to its opposite link type; the column assignment is not relevant, i.e., both link types are opposite to each other.

Table 8.4: Accountability Link Types.

Link Type	Opposite Link Type
CreatedBy	Created
ModifiedBy	Modified
RemovedBy	Removed
AppliedBy	Applied
DecomposedBy	Decomposed
HomologatedBy	Homologated
RejectedBy	Rejected

CreatedBy Link Type

Let β be an artifact and let a be an actor. Let e be a link starting in β and having a as its destination. The link e is a *CreatedBy* link type if it conveys the information that β was created by a.

Formalization. Let $\beta = \{i_1, \ldots, i_n\}$, $n \geq 1$, be an artifact and let a be an actor. Let $TS_k = (AC_k, SS_k, RS_k, PS_k, IEL_k)$ be a Traceability Space k. If $\beta \in SS_k$, $\beta \subseteq \{i_p, \ldots, i_q\}$, $1 \leq p \leq q \leq n$, then $\exists a \in AC_k : \{i_p, \ldots, i_q\}$ was created by a.

Created Link Type

Let a be an actor and let β be an artifact. Let e be a link starting in a and having β as its destination. The link e is a *Created* link type if it conveys the information that a created β.

The link type Created is an opposite link type to the CreatedBy link type.

Formalization. Let a be an actor and $\beta = \{i_1, \ldots, i_n\}$, $n \geq 1$, be an artifact. Let $TS_k = (AC_k, SS_k, RS_k, PS_k, IEL_k)$ be a Traceability Space k. If $a \in AC_k$, then $\exists \beta \in SS_k$, $\beta \subseteq \{i_p, \ldots, i_q\}$, $1 \leq p \leq q \leq n : a$ created $\{i_p, \ldots, i_q\}$.

ModifiedBy Link Type

Let β be an artifact and let a be an actor. Let e be a link starting in β and having a as its destination. The link e is a *ModifiedBy* link type if it conveys the information that β was modified by a.

Formalization. Let $\beta = \{i_1, \ldots, i_p\} \cup \{i'_q, \ldots, i'_n\}$, $1 < p < q \leq n$, be an artifact, $\{i_1, \ldots, i_p\}$ may be empty, and let a be an actor. Let $TS_k = (AC_k, SS_k, RS_k, PS_k, IEL_k)$ be a Traceability Space k. If $\beta \in SS_k$, $\{i'_q, \ldots, i'_n\} \subseteq \{i_r, \ldots, i_s\}$, $q \leq r \leq s \leq n$,

then $\exists a \in AC_k : \{i_r \ldots, i_s\}$ was modified by a.

Modified Link Type

Let a be an actor and let β be an artifact. Let e be a link starting in a and having β as its destination. The link e is a *Modified* link type if it conveys the information that a modified β.

The link type Modified is an opposite link type to the ModifiedBy link type.

Formalization. Let a be an actor and let $\beta = \{i_1, \ldots, i_p\} \cup \{i'_q, \ldots, i'_n\}$, $1 < p < q \leq n$, be an artifact, and $\{i_1, \ldots, i_p\}$ may be empty. Let $TS_k = (AC_k, SS_k, RS_k, PS_k, IEL_k)$ be a Traceability Space k. If $a \in AC_k$, then $\exists \beta \in SS_k$, $\{i'_q, \ldots, i'_n\} \subseteq \{i_r, \ldots, i_s\}$, $q \leq r \leq s \leq n$: a modified $\{i_r \ldots, i_s\}$.

RemovedBy Link Type

Let β be an artifact and let a be an actor. Let e be a link starting in β and having a as its destination. The link e is a *RemovedBy* link type if it conveys the information that β was removed by a.

Formalization. Let $\beta = \{i_1, \ldots, i_n\}$, $n \geq 1$, be an artifact, $\{j_q, \ldots, j_n\}$ be the set of information concerning β on other artifacts[1], and let a be an actor. Let $TS_k = (AC_k, SS_k, RS_k, PS_k, IEL_k)$ be a Traceability Space k. If $\beta \in SS_k$ and $\{j_q, \ldots, j_n\} \in SS_k$, $\{i_1, \ldots, i_n\} \cup \{j_q, \ldots, j_n\} \subseteq \{l_1, \ldots, l_p\}$, $p \geq 1$, then $\exists a \in AC_k : \{l_1, \ldots, l_p\}$ was removed by a.

Removed Link Type

Let a be an actor and let β be an artifact. Let e be a link starting in a and having β as its destination. The link e is a *Removed* link type if it conveys the information that a removed β.

The link type Removed is an opposite link type to the RemovedBy link type.

Formalization. Let a be an actor and let $\beta = \{i_1, \ldots, i_n\}$, $n \geq 1$, be an artifact, $\{j_q, \ldots, j_n\}$ be the set of information concerning β on other artifacts. Let $TS_k = (AC_k, SS_k, RS_k, PS_k, IEL_k)$ be a Traceability Space k. If $a \in AC_k$, then $\exists \beta \in SS_k$ and $\exists \{j_q, \ldots, j_n\} \in SS_k$, $\{i_1, \ldots, i_n\} \cup \{j_q, \ldots, j_n\} \subseteq \{l_1, \ldots, l_p\}$, $p \geq 1$: a removed $\{l_1, \ldots, l_p\}$.

AppliedBy Link Type

Let β be an artifact and let a be an actor. Let e be a link starting in β and having a as its destination. The link e is an *AppliedBy* link type if it conveys the information that β was applied by a.

[1] For instance, a call to a function from β.

Formalization. Let $\beta = \{i_1, \dots, i_n\}$, $n \geq 1$, be an artifact and let a be an actor. Let $TS_k = (AC_k,\ SS_k,\ RS_k,\ PS_k,\ IEL_k)$ be a Traceability Space k. If $\beta \in SS_k$, $\beta \subseteq \{i_p, \dots, i_q\}$, $1 \leq p \leq q \leq n$, then $\exists a \in AC_k : \{i_p, \dots, i_q\}$ was applied by a.

Applied Link Type

Let a be an actor and let β be an artifact. Let e be a link starting in a and having β as its destination. The link e is an *Applied* link type if it conveys the information that a applied β.

The link type Applied is an opposite link type to the AppliedBy link type.

Formalization. Let a be an actor and $\beta = \{i_1, \dots, i_n\}$, $n \geq 1$, be an artifact. Let $TS_k = (AC_k,\ SS_k,\ RS_k,\ PS_k,\ IEL_k)$ be a Traceability Space k. If $a \in AC_k$, then $\exists \beta \in SS_k$, $\beta \subseteq \{i_p, \dots, i_q\}$, $1 \leq p \leq q \leq n : a$ applied $\{i_p, \dots, i_q\}$.

DecomposedBy Link Type

Let β be an artifact and let a be an actor. Let e be a link starting in β and having a as its destination. The link e is a *DecomposedBy* link type if it conveys the information that β was decomposed by a.

Formalization. Let β be an artifact decomposed to the set of artifacts $\beta_1, \dots, \beta_n$. Let $\beta_1 \cup \ldots \cup \beta_n = \{i'_1, \dots, i'_m\}$, $m \geq 1$. Let a be an actor and let $TS_k = (AC_k,\ SS_k,\ RS_k,\ PS_k,\ IEL_k)$ be a Traceability Space k. If $\{i'_1, \dots, i'_m\} \in SS_k$, $\{i'_1, \dots, i'_m\} \subseteq \{i_p, \dots, i_q\}$, $1 \leq p \leq q \leq m$, then $\exists a \in AC_k : \{i_p, \dots, i_q\}$ was created by a.[2]

Decomposed Link Type

Let a be an actor and let β be an artifact. Let e be a link starting in a and having β as its destination. The link e is a *Decomposed* link type if it conveys the information that a decomposed β.

The link type Decomposed is an opposite link type to the DecomposedBy link type.

Formalization. Let a be an actor and let β be an artifact decomposed to the set of artifacts $\beta_1, \dots, \beta_n$. Let $\beta_1 \cup \ldots \cup \beta_n = \{i'_1, \dots, i'_m\}$, $m \geq 1$, and let $TS_k = (AC_k,\ SS_k,\ RS_k,\ PS_k,\ IEL_k)$ be a Traceability Space k. If $a \in AC_k$, then $\exists \{i'_1, \dots, i'_m\} \in SS_k$, $\{i'_1, \dots, i'_m\} \subseteq \{i_p, \dots, i_q\}$, $1 \leq p \leq q \leq m : a$ created $\{i_p, \dots, i_q\}$.

[2] There is no new relevant information added during a decomposition; however, it may be necessary to add structural data to create the new artifacts. For instance, suppose a requirement is broken into three different requirements; each one needs a new header. Thus, an actor which decomposes an artifact has to create new artifacts by using existing information and by adding and removing structural data.

HomologatedBy Link Type

Let β be an artifact and let a be an actor. Let e be a link starting in β and having a as its destination. The link e is a *HomologatedBy* link type if it conveys the information that β was homologated by a.

Formalization. Let β be an artifact and let a be an actor. Let A be a group of actors where $a \in A$. Let $TS_k = (AC_k,\ SS_k,\ RS_k,\ PS_k,\ IEL_k)$ be a Traceability Space k. If $\beta \in SS_k$, then $\exists A \in AC_k : \beta$ was homologated by A.

Homologated Link Type

Let a be an actor and let β be an artifact. Let e be a link starting in a and having β as its destination. The link e is a *Homologated* link type if it conveys the information that a homologated β.

The link type Homologated is an opposite link type to the HomologatedBy link type.

Formalization. Let a be an actor and β be an artifact. Let A be a group of actors where $a \in A$. Let $TS_k = (AC_k,\ SS_k,\ RS_k,\ PS_k,\ IEL_k)$ be a Traceability Space k. If $A \in AC_k$, then $\exists \beta \in SS_k : A$ homologated β.

RejectedBy Link Type

Let β be an artifact and let a be an actor. Let e be a link starting in β and having a as its destination. The link e is a *RejectedBy* link type if it conveys the information that β was rejected by a.

Formalization. Let β be an artifact and let a be an actor. Let A be a group of actors where $a \in A$. Let $TS_k = (AC_k,\ SS_k,\ RS_k,\ PS_k,\ IEL_k)$ be a Traceability Space k. If $\beta \in SS_k$, then $\exists A \in AC_k : \beta$ was rejected by A.

Rejected Link Type

Let a be an actor and let β be an artifact. Let e be a link starting in a and having β as its destination. The link e is a *Rejected* link type if it conveys the information that a rejected β.

The link type Rejected is an opposite link type to the RejectedBy link type.

Formalization. Let a be an actor and β be an artifact. Let A be a group of actors where $a \in A$. Let $TS_k = (AC_k,\ SS_k,\ RS_k,\ PS_k,\ IEL_k)$ be a Traceability Space k. If $A \in AC_k$, then $\exists \beta \in SS_k : A$ rejected β.

These fourteen link types fulfill the Accountability link type of the Reference Model by enabling the atribution of authorship to the basic actions of creation, modification, removal, application, decomposition, and homologation. The accountability for the action of homologation was divided into two pairs of opposite link types: HomologatedBy and

Homologated, and RejectedBy and Rejected; this enables the identification of the result of the action by link type.

8.2.4 Permission Link Types

The following link types belong to the Permission link type of the Reference Model: MayModify, MayBeModifiedBy, MayRemove, MayBeRemovedBy, MayApply, MayBeAppliedBy, MayDecompose, MayBeDecomposedBy, MayHomologate, MayBeHomologatedBy, MayActivate, and MayBeActivatedBy.

Table 8.5 matches each link type to its opposite link type; the column assignment is not relevant, i.e., both link types are opposite to each other.

Table 8.5: Permission Link Types.

Link Type	Opposite Link Type
MayModify	MayBeModifiedBy
MayRemove	MayBeRemovedBy
MayApply	MayBeAppliedBy
MayDecompose	MayBeDecomposedBy
MayHomologate	MayBeHomologatedBy
MayActivate	MayBeActivatedBy

MayModify Link Type

Let a be an actor and let β be an artifact. Let e be a link starting in a and having β as its destination. The link e is a *MayModify* link type if it conveys the information that a is allowed to modify β.

Formalization. Let a be an actor and let β be an artifact. Let $TS_k = (AC_k, SS_k, RS_k, PS_k, IEL_k)$ be a Traceability Space k. If $a \in AC_k$, then $(\exists \beta \in SS_k) \wedge (\exists r_i \in RS_k) : r_i$ allows a to modify β.

MayBeModifiedBy Link Type

Let β be an artifact and let a be an actor. Let e be a link starting in β and having a as its destination. The link e is a *MayBeModifiedBy* link type if it conveys the information that β may be modified by a.

The link type MayBeModifiedBy is an opposite link type to the MayModify link type.

Formalization. Let β be an artifact and let a be an actor. Let $TS_k = (AC_k, SS_k, RS_k, PS_k, IEL_k)$ be a Traceability Space k. If $\beta \in SS_k$, then $(\exists a \in AC_k) \wedge (\exists r_i \in RS_k) : r_i$

allows a to modify β.

MayRemove Link Type

Let a be an actor and let β be an artifact. Let e be a link starting in a and having β as its destination. The link e is a *MayRemove* link type if it conveys the information that a is allowed to remove β.

Formalization. Let a be an actor and let β be an artifact. Let $TS_k = (AC_k, SS_k, RS_k, PS_k, IEL_k)$ be a Traceability Space k. If $a \in AC_k$, then $(\exists \beta \in SS_k) \wedge (\exists r_i \in RS_k) : r_i$ allows a to remove β.

MayBeRemovedBy Link Type

Let β be an artifact and let a be an actor. Let e be a link starting in β and having a as its destination. The link e is a *MayBeRemovedBy* link type if it conveys the information that β may be removed by a.

The link type MayBeRemovedBy is an opposite link type to the MayRemove link type.

Formalization. Let β be an artifact and let a be an actor. Let $TS_k = (AC_k, SS_k, RS_k, PS_k, IEL_k)$ be a Traceability Space k. If $\beta \in SS_k$, then $(\exists a \in AC_k) \wedge (\exists r_i \in RS_k) : r_i$ allows a to remove β.

MayApply Link Type

Let a be an actor and let β be an artifact. Let e be a link starting in a and having β as its destination. The link e is a *MayApply* link type if it conveys the information that a is allowed to apply β.

Formalization. Let a be an actor and let β be an artifact. Let $TS_k = (AC_k, SS_k, RS_k, PS_k, IEL_k)$ be a Traceability Space k. If $a \in AC_k$, then $(\exists \beta \in SS_k) \wedge (\exists r_i \in RS_k) : r_i$ allows a to apply β.

MayBeAppliedBy Link Type

Let β be an artifact and let a be an actor. Let e be a link starting in β and having a as its destination. The link e is a *MayBeAppliedBy* link type if it conveys the information that β may be applied by a.

The link type MayBeAppliedBy is an opposite link type to the MayApply link type.

Formalization. Let β be an artifact and let a be an actor. Let $TS_k = (AC_k, SS_k, RS_k, PS_k, IEL_k)$ be a Traceability Space k. If $\beta \in SS_k$, then $(\exists a \in AC_k) \wedge (\exists r_i \in RS_k) : r_i$ allows a to apply β.

MayDecompose Link Type

Let a be an actor and let β be an artifact. Let e be a link starting in a and having β as its destination. The link e is a *MayDecompose* link type if it conveys the information that a is allowed to decompose β.

Formalization. Let a be an actor and let β be an artifact. Let $TS_k = (AC_k, SS_k, RS_k, PS_k, IEL_k)$ be a Traceability Space k. If $a \in AC_k$, then $(\exists \beta \in SS_k) \wedge (\exists r_i \in RS_k) : r_i$ allows a to decompose β.

MayBeDecomposedBy Link Type

Let β be an artifact and let a be an actor. Let e be a link starting in β and having a as its destination. The link e is a *MayBeDecomposedBy* link type if it conveys the information that β may be decomposed by a.

The link type MayBeDecomposedBy is an opposite link type to the MayDecompose link type.

Formalization. Let β be an artifact and let a be an actor. Let $TS_k = (AC_k, SS_k, RS_k, PS_k, IEL_k)$ be a Traceability Space k. If $\beta \in SS_k$, then $(\exists a \in AC_k) \wedge (\exists r_i \in RS_k) : r_i$ allows a to decompose β.

MayHomologate Link Type

Let a be an actor and let β be an artifact. Let e be a link starting in a and having β as its destination. The link e is a *MayHomologate* link type if it conveys the information that a is allowed to homologate β.[3]

Formalization. Let a be an actor and let β be an artifact. Let $TS_k = (AC_k, SS_k, RS_k, PS_k, IEL_k)$ be a Traceability Space k. If $a \in AC_k$, then $(\exists \beta \in SS_k) \wedge (\exists r_i \in RS_k) : r_i$ allows a to participate in the process of homologation of β.

MayBeHomologatedBy Link Type

Let β be an artifact and let a be an actor. Let e be a link starting in β and having a as its destination. The link e is a *MayBeHomologatedBy* link type if it conveys the information that β may be homologated by a.

The link type MayBeHomologatedBy is an opposite link type to the MayHomologate link type.

Formalization. Let β be an artifact and let a be an actor. Let $TS_k = (AC_k, SS_k, RS_k, PS_k, IEL_k)$ be a Traceability Space k. If $\beta \in SS_k$, then $(\exists a \in AC_k) \wedge (\exists r_i \in RS_k) : r_i$ allows a to participate in the process of homologation of β.

---

[3] "To homologate β" is used in the sense of participating in the homologation process of β, instead of automatically adding β to the Active System.

MayActivate Link Type

Let a be an actor and let β be an artifact. Let e be a link starting in a and having β as its destination. The link e is a *MayActivate* link type if it conveys the information that a is allowed to activate β.[4]

Formalization. Let a be an actor and let β be an artifact. Let $TS_k = (AC_k, SS_k, RS_k, PS_k, IEL_k)$ be a Traceability Space k. If $a \in AC_k$, then $(\exists \beta \in SS_k) \wedge (\exists r_i \in RS_k) : r_i$ allows a to participate in the process of activation of β.

MayBeActivatedBy Link Type

Let β be an artifact and let a be an actor. Let e be a link starting in β and having a as its destination. The link e is a *MayBeActivatedBy* link type if it conveys the information that β may be activated by a.

The link type MayBeActivatedBy is an opposite link type to the MayActivate link type.

Formalization. Let β be an artifact and let a be an actor. Let $TS_k = (AC_k, SS_k, RS_k, PS_k, IEL_k)$ be a Traceability Space k. If $\beta \in SS_k$, then $(\exists a \in AC_k) \wedge (\exists r_i \in RS_k) : r_i$ allows a to participate in the process of activation of β.

Permission links connecting actors in the Actors Space to artifacts in the System Space are generated by the application of rules from the Rules Space. For instance, a *MayModify* link connecting an actor a to an artifact β of type A_1 may have been generated by the application of a rule defining that a may modify artifacts of type A_1; links may be also generated manually. There are rules in the Rules Space which do not generate permission links connecting actors to artifacts; for instance, there may be a rule defining that an actor a may create artifacts of type A_1 but it does not generate a corresponding permission link.

These twelve link types fulfill the Permission link type of the Reference Model by conveying information on authorization to perform the basic actions of creation, modification, removal, application, decomposition, and homologation, plus the activation action. The activation action is the addition of inactive artifacts which were previously removed or rejected during homologation, to the Active System (see Chapter 9 for more details).

8.2.5 Characterize Action Link Types

The following link types belong to the Characterize Action link type of the Reference Model: DefinesModificationOf, ModificationDefinedBy, DefinesDecompositionOf, DecompositionDefinedBy, DefinesRemovalOf, RemovalDefinedBy, DefinesActivationOf, ActivationDefinedBy, DefinesCreationOf, CreationDefinedBy, SubjectToApplicationOf, AppliesTo, JustifiesHomologationOf, HomologationJustifiedBy, JustifiesRejectionOf, RejectionJustifiedBy, JustifiesApplicationOf, and ApplicationJustifiedBy.

[4]"To activate β" is used in the sense of participating in the activation process of β.

Table 8.6 matches each link type to its opposite link type; the column assignment is not relevant, i.e., both link types are opposite to each other.

Table 8.6: Characterize Action Link Types.

Link Type	**Opposite Link Type**
DefinesModificationOf	ModificationDefinedBy
DefinesDecompositionOf	DecompositionDefinedBy
DefinesRemovalOf	RemovalDefinedBy
DefinesActivationOf	ActivationDefinedBy
DefinesCreationOf	CreationDefinedBy
SubjectToApplicationOf	AppliesTo
JustifiesHomologationOf	HomologationJustifiedBy
JustifiesRejectionOf	RejectionJustifiedBy
JustifiesApplicationOf	ApplicationJustifiedBy

Each pair of opposite Characterize Action link types may convey three types of information concerning an action: justification and description, solely justification, or solely description. These link types may convey up to two types of information, given the content of the related Rationale Artifact. Table 8.7 shows the link types organized by the type of information they convey. The first column shows the information conveyed; the second column shows the link types.

Table 8.7: Characterize Action Link Types by Information Conveyed.

Information Conveyed	**Link Types**
Justification and Description	DefinesModificationOf, ModificationDefinedBy, DefinesDecompositionOf, DecompositionDefinedBy
Justification, or Justification and Description	DefinesRemovalOf, RemovalDefinedBy, DefinesActivationOf, ActivationDefinedBy, DefinesCreationOf, CreationDefinedBy
Description	SubjectToApplicationOf, AppliesTo
Justification	JustifiesHomologationOf, HomologationJustifiedBy, JustifiesRejectionOf, RejectionJustifiedBy, JustifiesApplicationOf, ApplicationJustifiedBy

DefinesModificationOf Link Type

Let α and β be artifacts where α is a Rationale for Modification; let e be a link starting in α and having β as its destination. The link e is a *DefinesModificationOf* link type if it conveys the information that β is the target of α, and α justifies and describes a modification of β.

Formalization. Let J_{mod}: $(x, y) \rightarrow \{True, False\}$ be a predicate, which given two sets of blocks of information as arguments, returns True if the first set justifies the modification of the second set and returns False otherwise. Let D_{mod}: $(x, y) \rightarrow \{True, False\}$ be a predicate, which given two sets of blocks of information as arguments, returns True if the first set describes the modification of the second set and returns False otherwise. Let $\alpha = \{i_1, \ldots, i_p\} \cup \{i_q, \ldots, i_n\}$, $1 \leq p \leq q \leq n$, and β be artifacts. Let SS_k be a System Space k. If $\alpha \in SS_k$, then $\exists \beta \in SS_k$: $((J_{mod}(\{i_1, \ldots, i_p\}, \beta) =$ True$) \wedge (D_{mod}(\{i_q, \ldots, i_n\}, \beta) =$ True$))$.

ModificationDefinedBy Link Type

Let α and β be artifacts where α is a Rationale for Modification; let e be a link starting in β and having α as its destination. The link e is a *ModificationDefinedBy* link type if it conveys the information that α targets β, and β has its modification justified and described by α.

The link type ModificationDefinedBy is an opposite link type to the DefinesModificationOf link type.

Formalization. Let $\alpha = \{i_1, \ldots, i_p\} \cup \{i_q, \ldots, i_n\}$, $1 \leq p \leq q \leq n$, and β be artifacts. Let SS_k be a System Space k. If $\beta \in SS_k$, then $\exists \alpha \in SS_k$: $((J_{mod}(\{i_1, \ldots, i_p\}, \beta) =$ True$) \wedge (D_{mod}(\{i_q, \ldots, i_n\}, \beta) =$ True$))$.

DefinesDecompositionOf Link Type

Let α and β be artifacts where α is a Rationale for Decomposition; let e be a link starting in α and having β as its destination. The link e is a *DefinesDecompositionOf* link type if it conveys the information that β is the target of α, and α justifies and describes a decomposition of β.

Formalization. Let J_{dec}: $(x, y) \rightarrow \{True, False\}$ be a predicate, which given two sets of blocks of information as arguments, returns True if the first set justifies the decomposition of the second set and returns False otherwise. Let D_{dec}: $(x, y) \rightarrow \{True, False\}$ be a predicate, which given two sets of blocks of information as arguments, returns True if the first set describes the decomposition of the second set and returns False otherwise. Let $\alpha = \{i_1, \ldots, i_p\} \cup \{i_q, \ldots, i_n\}$, $1 \leq p \leq q \leq n$, and β be artifacts. Let SS_k be a System Space k. If $\alpha \in SS_k$, then $\exists \beta \in SS_k$: $((J_{dec}(\{i_1, \ldots, i_p\}, \beta) =$ True$) \wedge (D_{dec}(\{i_q, \ldots, i_n\}, \beta) =$ True$))$.

DecompositionDefinedBy Link Type

Let α and β be artifacts where α is a Rationale for Decomposition; let e be a link starting in β and having α as its destination. The link e is a *DecompositionDefinedBy* link type if it conveys the information that α targets β, and β has its decomposition justified and described by α.

The link type DecompositionDefinedBy is an opposite link type to the DefinesDecompositionOf link type.

Formalization. Let $\alpha = \{i_1, \ldots, i_p\} \cup \{i_q, \ldots, i_n\}$, $1 \leq p \leq q \leq n$, and β be artifacts. Let SS_k be a System Space k. If $\beta \in SS_k$, then $\exists \alpha \in SS_k : ((J_{dec}(\{i_1, \ldots, i_p\},\beta) = \text{True}) \wedge (D_{dec}(\{i_q, \ldots, i_n\},\beta) = \text{True}))$.

The link types *DefinesRemovalOf* and *RemovalDefinedBy* are different to the above link types since they have two possible roles: they may only justify, or justify and describe, an action.

DefinesRemovalOf Link Type

Let α and β be artifacts where α is a Rationale for Removal; let e be a link starting in α and having β as its destination. The link e is a *DefinesRemovalOf* link type if it conveys the that β is the target of α, and α justifies, or justifies and describes, a removal of β.

Formalization. Let J_{rem}: $(x, y) \rightarrow \{True, False\}$ be a predicate, which given two sets of blocks of information as arguments, returns True if the first set justifies the removal of the second set and returns False otherwise. Let D_{rem}: $(x, y) \rightarrow \{True, False\}$ be a predicate, which given two sets of blocks of information as arguments, returns True if the first set describes the removal of the second set and returns False otherwise. Let $\alpha = \{i_1, \ldots, i_p\} \cup \{i_q, \ldots, i_n\}$, $1 \leq p \leq q \leq n$, and β be artifacts. Let SS_k be a System Space k. If $\alpha \in SS_k$, then $\exists \beta \in SS_k : ((J_{rem}(\{i_1, \ldots, i_p\},\beta) = \text{True}) \wedge (\{i_q, \ldots, i_n\} = \emptyset)) \vee ((J_{rem}(\{i_1, \ldots, i_p\},\beta) = \text{True}) \wedge (D_{rem}(\{i_q, \ldots, i_n\},\beta) = \text{True}))$.

RemovalDefinedBy Link Type

Let α and β be artifacts where α is a Rationale for Removal; let e be a link starting in β and having α as its destination. The link e is a *RemovalDefinedBy* link type if it conveys the information that α targets β, and β has its removal justified, or justified and described, by α.

The link type RemovalDefinedBy is an opposite link type to the DefinesRemovalOf link type.

Formalization. Let $\alpha = \{i_1, \ldots, i_p\} \cup \{i_q, \ldots, i_n\}$, $1 \leq p \leq q \leq n$, and β be artifacts. Let SS_k be a System Space k. If $\beta \in SS_k$, then $\exists \alpha \in SS_k : ((J_{rem}(\{i_1, \ldots, i_p\},\beta) = \text{True}) \wedge (\{i_q, \ldots, i_n\} = \emptyset)) \vee ((J_{rem}(\{i_1, \ldots, i_p\},\beta) = \text{True}) \wedge (D_{rem}(\{i_q, \ldots, i_n\},\beta) = \text{True}))$.

DefinesActivationOf Link Type

Let α and β be artifacts where α is a Rationale for Activation; let e be a link starting in α and having β as its destination. The link e is a *DefinesActivationOf* link type if it conveys the information that β is the target of α, and α justifies, or justifies and describes, the activation of β.

Formalization. Let J_{act}: $(x, y) \rightarrow \{True, False\}$ be a predicate, which given two sets of blocks of information as arguments, returns True if the first set justifies the activation of the second set and returns False otherwise. Let D_{act}: $(x, y) \rightarrow \{True, False\}$ be a predicate, which given two sets of blocks of information as arguments, returns True if the first set describes the activation of the second set and returns False otherwise. Let $\alpha = \{i_1, \ldots, i_p\} \cup \{i_q, \ldots, i_n\}$, $1 \leq p \leq q \leq n$, and β be artifacts. Let SS_k be a System Space k. If $\alpha \in SS_k$, then $\exists \beta \in SS_k : ((J_{act}(\{i_1, \ldots, i_p\},\beta) = \text{True}) \wedge (\{i_q, \ldots, i_n\} = \emptyset)) \vee ((J_{act}(\{i_1, \ldots, i_p\},\beta) = \text{True}) \wedge (D_{act}(\{i_q, \ldots, i_n\},\beta) = \text{True}))$.

ActivationDefinedBy Link Type

Let α and β be artifacts where α is a Rationale for Activation; let e be a link starting in β and having α as its destination. The link e is an *ActivationDefinedBy* link type if it conveys the information that α targets β, and β has its activation justified, or justified and described, by α.

The link type ActivationDefinedBy is an opposite link type to the DefinesActivationOf link type.

Formalization. Let $\alpha = \{i_1, \ldots, i_p\} \cup \{i_q, \ldots, i_n\}$, $1 \leq p \leq q \leq n$, and β be artifacts. Let SS_k be a System Space k. If $\beta \in SS_k$, then $\exists \alpha \in SS_k : ((J_{act}(\{i_1, \ldots, i_p\},\beta) = \text{True}) \wedge (\{i_q, \ldots, i_n\} = \emptyset)) \vee ((J_{act}(\{i_1, \ldots, i_p\},\beta) = \text{True}) \wedge (D_{act}(\{i_q, \ldots, i_n\},\beta) = \text{True}))$.

DefinesCreationOf Link Type

Let α and β be artifacts where α is a Rationale for Creation; let e be a link starting in α and having β as its destination. The link e is a *DefinesCreationOf* link type if it conveys the that β is the target of α, and α justifies, or justifies and describes, the creation of β.

Formalization. Let J_{cre}: $(x, y) \rightarrow \{True, False\}$ be a predicate, which given two sets of blocks of information as arguments, returns True if the first set justifies the creation of the second set and returns False otherwise. Let D_{cre}: $(x, y) \rightarrow \{True, False\}$ be a predicate, which given two sets of blocks of information as arguments, returns True if the first set describes the creation of the second set and returns False otherwise. Let $\alpha = \{i_1, \ldots, i_p\} \cup \{i_q, \ldots, i_n\}$, $1 \leq p \leq q \leq n$, and β be artifacts. Let SS_k be a System Space k. If $\alpha \in SS_k$, then $\exists \beta \in SS_k : ((J_{cre}(\{i_1, \ldots, i_p\},\beta) = \text{True}) \wedge (\{i_q, \ldots, i_n\} = \emptyset)) \vee ((J_{cre}(\{i_1, \ldots, i_p\},\beta) = \text{True}) \wedge (D_{cre}(\{i_q, \ldots, i_n\},\beta) = \text{True}))$.

CreationDefinedBy Link Type

Let α and β be artifacts where α is a Rationale for Creation; let e be a link starting in β and having α as its destination. The link e is a *CreationDefinedBy* link type if it conveys the information that α targets β, and β has its creation justified, or justified and described, by α.

The link type CreationDefinedBy is an opposite link type to the DefinesCreationOf link type.

Formalization. Let $\alpha = \{i_1, \ldots, i_p\} \cup \{i_q, \ldots, i_n\}$, $1 \leq p \leq q \leq n$, and β be artifacts. Let SS_k be a System Space k. If $\beta \in SS_k$, then $\exists \alpha \in SS_k : ((J_{cre}(\{i_1, \ldots, i_p\},\beta) = \text{True}) \wedge (\{i_q, \ldots, i_n\} = \emptyset)) \vee ((J_{cre}(\{i_1, \ldots, i_p\},\beta) = \text{True}) \wedge (D_{cre}(\{i_q, \ldots, i_n\},\beta) = \text{True}))$.

The link types DefinesCreationOf and CreationDefinedBy are optional[5], not being considered by the processes in Chapter 9; these are provided to enable users to justify, or justify and describe, the creation of artifacts. For instance, an user who creates an artifact may choose to create a Rationale for Creation explaining why it is needed; or, the user may create the Rationale for Creation justifying and describing the artifact. In this case, if the Rationale for Creation is homologated, the artifact will be created given the instructions in the Rationale for Creation. In both cases, the artifact will be connected to the Rationale by DefinesCreationOf and CreationDefinedBy link types.

The link types AppliesTo and SubjectToApplicationOf are different to the above link types since they only describe an action; they do not connect an artifact to a Rationale, but two non-rationale artifacts, one which should be applied onto another.

AppliesTo Link Type

Let α and β be artifacts. Let e be a link starting in α and having β as its destination. The link e is an *AppliesTo* link type if it conveys the information that α describes the application of α on β.

For instance, a test case describes how it should be applied to a piece of code.

Formalization. Let D_{apl}: $(x, y) \rightarrow \{True, False\}$ be a predicate, which given two sets of blocks of information as arguments, returns True if the first set describes its application on the second set and returns False otherwise. Let α and β be artifacts. Let SS_k be a System Space k. If $\alpha \in SS_k$, then $\exists \beta \in SS_k : D_{apl}(\alpha,\beta) = \text{True}$.

SubjectToApplicationOf Link Type

Let α and β be artifacts. Let e be a link starting in α and having β as its destination. The link e is an *SubjectToApplicationOf* link type if it conveys the information that α is subjected to the application of β, which describes its own application.

[5]Every link type is optional, given that an user may choose to customize the Metamodel as desired; however, these link types are provided as useful options.

The link type SubjectToApplicationOf is an opposite link type to the AppliesTo link type.

Formalization. Let α and β be artifacts. Let SS_k be a System Space k. If $\beta \in SS_k$, then $\exists \alpha \in SS_k : D_{apl}(\alpha,\beta) =$ True.

The link types JustifiesHomologationOf and JustifiesHomologationOf are different to the above link types since they only justify an action.

JustifiesHomologationOf Link Type

Let α and β be artifacts where α is a Rationale for Homologation or Rejection; let e be a link starting in α and having β as its destination. The link e is a *JustifiesHomologationOf* link type if it conveys the information that β is the target of α, and α justifies the homologation of β.

Formalization. Let J_{hom}: $(x, y) \rightarrow \{True, False\}$ be a predicate, which given two sets of blocks of information as arguments, returns True if the first set justifies the homologation of the second set and returns False otherwise. Let α and β be artifacts. Let SS_k be a System Space k. If $\alpha \in SS_k$, then $\exists \beta \in SS_k : J_{hom}$:$(\alpha,\beta) =$ True.

HomologationJustifiedBy Link Type

Let α and β be artifacts where α is a Rationale for Homologation or Rejection; let e be a link starting in β and having α as its destination. The link e is a *HomologationJustifiedBy* link type if it conveys the information that α targets β, and β has its homologation justified by α.

The link type HomologationJustifiedBy is an opposite link type to the JustifiesHomologationOf link type.

Formalization. Let α and β be artifacts. Let SS_k be a System Space k. If $\beta \in SS_k$, then $\exists \alpha \in SS_k : J_{hom}$:$(\alpha,\beta) =$ True.

JustifiesRejectionOf Link Type

Let α and β be artifacts where α is a Rationale for Homologation or Rejection; let e be a link starting in α and having β as its destination. The link e is a *JustifiesRejectionOf* link type if it conveys the information that β is the target of α, and α justifies the rejection of β.

Formalization. Let J_{rej}: $(x, y) \rightarrow \{True, False\}$ be a predicate, which given two sets of blocks of information as arguments, returns True if the first set justifies the rejection of the second set and returns False otherwise. Let α and β be artifacts. Let SS_k be a System Space k. If $\alpha \in SS_k$, then $\exists \beta \in SS_k : J_{rej}$:$(\alpha,\beta) =$ True.

RejectionJustifiedBy Link Type

Let α and β be artifacts where α is a Rationale for Homologation or Rejection; let e be a link starting in β and having α as its destination. The link e is a *RejectionJustifiedBy* link type if it conveys the information that α targets β, and β has its rejection justified by α.

The link type RejectionJustifiedBy is an opposite link type to the JustifiesRejectionOf link type.

Formalization. Let α and β be artifacts. Let SS_k be a System Space k. If $\beta \in SS_k$, then $\exists\alpha \in SS_k : J_{rej}$:($\alpha$,$\beta$) = True.

JustifiesApplicationOf Link Type

Let α and β be artifacts where α is a Rationale for Application; let e be a link starting in α and having β as its destination. The link e is a *JustifiesApplicationOf* link type if it conveys the information that β is the target of α, and α justifies the application of β.

Formalization. Let J_{apl}: $(x, y) \rightarrow \{True, False\}$ be a predicate, which given two sets of blocks of information as arguments, returns True if the first set justifies the application of the second set and returns False otherwise. Let α and β be artifacts. Let SS_k be a System Space k. If $\alpha \in SS_k$, then $\exists\beta \in SS_k : J_{apl}$:($\alpha$,$\beta$) = True.

ApplicationJustifiedBy Link Type

Let α and β be artifacts where α is a Rationale for Application; let e be a link starting in β and having α as its destination. The link e is an *ApplicationJustifiedBy* link type if it conveys the information that α targets β, and β has its application justified by α.

The link type ApplicationJustifiedBy is an opposite link type to the JustifiesApplicationOf link type.

Formalization. Let α and β be artifacts. Let SS_k be a System Space k. If $\beta \in SS_k$, then $\exists\alpha \in SS_k : J_{apl}$:($\alpha$,$\beta$) = True.

The link types *JustifiesApplicationOf* and *ApplicationJustifiedBy* are optional; these are provided so that every basic action has a corresponding Rationale artifact. These link types may be utilized if an user needs to justify the application of artifacts.

These eighteen link types (including four optional link types) fulfill the Characterize Action link type of the Reference Model by conveying the rationale and/or description for the basic actions of creation, modification, removal, application, decomposition, and homologation, plus the activation action.

8.2.6 Action Outcome Link Types

The following link types belong to the Action Outcome link type of the Reference Model: ApplicationProduced, and ProducedByApplicationOf.

Table 8.8 matches each link type to its opposite link type; the column assignment is not relevant, i.e., both link types are opposite to each other.

Table 8.8: Action Outcome Link Types.

Link Type	Opposite Link Type
ApplicationProduced	ProducedByApplicationOf

ApplicationProduced Link Type

Let α and β be artifacts. Let e be a link starting in α and having β as its destination. The link e is an *ApplicationProduced* link type if it conveys the information that the application of α on another artifact resulted in the creation of β.

Formalization. Let *Apply*: $(x, y) \rightarrow z$ be the function which, given two artifacts, applies the first artifact on the second artifact and returns a resulting artifact. Let α, β, and γ be artifacts. Let SS_k be a System Space k. If $((\alpha \in SS_k) \wedge (\beta \in SS_k))$, then $\exists \gamma \in SS_k$: $Apply(\alpha, \beta) = \gamma$.

ProducedByApplicationOf Link Type

Let α and β be artifacts. Let e be a link starting in β and having α as its destination. The link e is an *ProducedByApplicationOf* link type if it conveys the information that β was created as the result of the application of α on another artifact.

The link type ProducedByApplicationOf is an opposite link type to the Application-Produced link type.

Formalization. Let α, β, and γ be artifacts. Let SS_k be a System Space k. If $\gamma \in SS_k$, then $(\exists \alpha \in SS_k) \wedge (\exists \beta \in SS_k)$: $Apply(\alpha, \beta) = \gamma$.

These two link types fulfill the Action Outcome link type of the Reference Model by conveying the outcome of the action of application; it is not as common to have outcome artifacts created as the result of the remaining basic actions. If it is necessary to model the creation of outcome artifacts given other actions, new link types may be created accordingly; for instance, if an user needs to have an artifact created as the outcome of an activation action, a pair of opposite link types should be created to connect the action starter, or the action target, to an outcome artifact.

8.2.7 Composition Link Types

The following link types belong to the Composition link type of the Reference Model: DecomposedFrom, DecomposedTo, PartOf, and ComprisedOf.

Table 8.9 matches each link type to its opposite link type; the column assignment is not relevant, i.e., both link types are opposite to each other.

Table 8.9: Composition Link Types.

Link Type	Opposite Link Type
DecomposedFrom	DecomposedTo
PartOf	ComprisedOf

DecomposedFrom Link Type

Let α and β be two artifacts and let e be a link starting in α and having β as its destination. The link e is a *DecomposedFrom* link type if it conveys the information that α was decomposed from β. i.e., the artifact β was decomposed to two or more artifacts, and α is one of these artifacts.

Formalization. Let Dec: $(\gamma, \{l_1, \ldots, l_p\}) \rightarrow \{\gamma_1, \ldots, \gamma_n\}$ be the function which, given two sets of blocks of information, returns a decomposition of the first set, according to instructions from the second set. Let α and β be artifacts, where $|\beta| > |\alpha|$. Let $\{l_1, \ldots, l_p\}$, $p \geq 1$, be a set of blocks of information describing a decomposition. Let $Dec(\beta, \{l_1, \ldots, l_p\}) = \{\beta_1, \ldots, \beta_n\}$. Let SS_k be a System Space k. If $\alpha \in SS_k$, then $(\exists \beta \in SS_k) \wedge (\exists \{l_1, \ldots, l_p\} \in SS_k) : \exists i, \beta_i = \alpha$.

DecomposedTo Link Type

Let α and β be two artifacts and let e be a link starting in β and having α as its destination. The link e is a *DecomposedTo* link type if it conveys the information that β was decomposed to α.

The link type DecomposedTo is an opposite link type to the DecomposedFrom link type.

Formalization. Let β and α be artifacts, where $|\beta| > |\alpha|$. Let $\{l_1, \ldots, l_p\}$, $p \geq 1$, be a set of blocks of information describing a decomposition. Let $Dec(\beta, \{l_1, \ldots, l_p\}) = \{\beta_1, \ldots, \beta_n\}$. Let SS_k be a System Space k. If $\beta \in SS_k$, then $(\exists \alpha \in SS_k) \wedge (\exists \{l_1, \ldots, l_p\} \in SS_k) : \exists i$, $\beta_i = \alpha$.

Definition 8.2.7. Let α be an artifact where $\alpha = \{\alpha_1, \alpha_2, \ldots, \alpha_n\}$, $n \geq 1$; each α_i, $1 \leq i \leq n$, is part of α; conversely, the artifact α is comprised of α_i.

PartOf Link Type

Let α and β be two artifacts and let e be a link starting in α and having β as its destination. The link e is a *PartOf* link type if it conveys the information that α is part of β.

Formalization. Let $\alpha = \{i_1, \ldots, i_n\}$, $n \geq 1$, and β be artifacts. Let SS_k be a System Space k. If $\alpha \in SS_k$, then $\exists \beta \in SS_k : \forall j$, $i_j \in \beta$.

ComprisedOf Link Type

Let α and β be two artifacts and let e be a link starting in β and having α as its destination. The link e is a *ComprisedOf* link type if it conveys the information that β is comprised of α.

The link type ComprisedOf is an opposite link type to the PartOf link type.

Formalization. Let β and $\alpha = \{i_1, \ldots, i_n\}$, $n \geq 1$, be artifacts. Let SS_k be a System Space k. If $\beta \in SS_k$, then $\exists \alpha \in SS_k : \forall j,\, i_j \in \beta$.

PartOf and ComprisedOf link types are useful to increase the level of detail of traceability relations. For instance, given two C++ classes having a dependency relation between themselves, these relations enable the identification of the specific method which fulfills the dependency.

These link types are also useful when having complex artifacts which, for some reason, may not be broken into smaller artifacts. For instance, suppose an artifact α is a set of test cases. One of these test cases is applied to a code fragment, resulting in the creation of a test log connecting to α. If α is classified into parts by using the PartOf link type and the ComprisedOf link type, it is possible to identify which specific test case resulted in the test log.

These four link types fulfill the Composition link type of the Reference Model by conveying the information that an artifact was decomposed to two or more artifacts, or that an artifact is part of another artifact.

8.3 Closing Remarks

The proposed link types fulfill the roles of the seven link types of the Reference Model. These are comprehensive, specific, and provide artifact coverage in the context of common development projects.

Some link types may be changed to become more specific, if necessary. For instance, each link type of the Characterize Action which is able to convey justification or justification and description may be split into two link types: a link type conveying justification, and a link type conveying justification and description. This change may result in changes to the related artifact types; i.e., splitting the corresponding artifact types. It is possible to keep the artifact types unchanged; thus, their specifity will not match the link type specifity.

Certain Constraint link types may also be split into more specific link types. There are several types of dependency relations which may be modeled by traceability link types; for instance, existential dependencies, dependencies on previous actions, etc.

There are also specific contexts which may demand additional link types; e.g., a safety-focused development project. However, this Metamodel is intended for general use, and we consider the proposed link types to be useful for common and domain-specific development projects alike.

Chapter 9

Processes for Traceability

The purpose of a traceability process is to maintain traceability consistency and system consistency. It achieves this purpose by prescribing a sequence of steps to be taken in different situations. These steps may be composed of: generation, reallocation, or deletion of traceability links; creation, modification, decomposition, homologation, application, activation, or removal of artifacts; and defining the characteristics of sets of actors needed to perform certain actions.

Furthermore, processes are essential for assessing the impact of actions performed in the System Space; the evaluation of actions before their implementation enables the possibility of choosing to avoid an action, or to replace it by a less impactful choice; e.g., the modification of an artifact may result in the generation of conflicts which will be more costly to solve than, for example, creating a new artifact which achieves the desired goal without the generation of the same issues. This is possible since the processes describe the necessary steps given each particular issue which may arise given an action, enabling the user to make an informed decision before implementing it. Also, processes offer general guidelines to be followed, being highly adaptable to particular cases.

We have created 7 processes satisfying the basic processes required by the Reference Model plus a non-basic process; these are: the Homologation Process, the Modification Process, the Decomposition Process, the Creation Process, the Removal Process, the Application Process, and the Activation Process. These cover all actions contemplated by the Metamodel; the action of reification is addressed by the Creation Process. Each process is defined in two ways: as an algorithm written in pseudocode similar to C, and as a detailed textual description; the algorithm provides the instructions to be followed, and the textual description provides the instructions to be followed plus related commentary.

Each process combines an action and the steps necessary to ensure consistency given this action; for instance, the Modification Process is composed of the action of modification and the necessary steps to avoid inconsistencies after the modification is performed. Every process uses the Homologation Process to approve artifacts; therefore traceability and system consistency may be ensured completely, or partially, by this process. To exemplify: the Removal Process performs only the action of removal, leaving the consistency management to the Homologation Process; on the other hand, the Decomposition Process uses the Homologation Process but also provides instructions to ensure traceability consistency given the new artifacts created during the process.

The homologation of modifications and removals may cause significant issues in the project; to manage these issues, a process to be used of together with the Homologation Process is provided. This process enables assessing the impact of homologating modifications and removals by identifying possible issues and providing matching solutions. Its benefits are twofold: (i) it enables the maintenance of traceability and system consistency if the action is performed, and (ii) it enables the user to make an informed decision on going through the action or not, given the corresponding consequences.

The processes are shown next; a general explanation of the process, an algorithm, and a textual description are provided. Observations on permissions and processes, and discussions on change impact analysis and the automatization of processes, are also provided.

Whenever checking for the existence of traceability links, only links connecting to artifacts in the Active System are considered; inactive artifacts do not cause consistency issues. Thus, every traceability link checked by a process is assumed to connect to an artifact in the Active System.

Partial links should be treated as non-partial links if they are the only intra-system partial links in the Active System concerning a certain dependency; e.g., if a dependency in the Active System is satisfied by two artifacts, and one of these artifacts is moved to the Inactive System, the partial link connecting the remaining artifact in the Active System and the dependent artifact should be treated as if it was a non-partial link. Also, transformations between non-partial and partial links are assumed to be done whenever applicable.

This chapter is structured as follows: Sections 9.1–9.7 describe each process of the Metamodel; Section 9.8 summarizes necessary updates to links in a project, given certain processes; and Section 9.9 discusses how processes relate to change impact analysis.

9.1 Homologation Process

Every newly created artifact starts off inactive; it has to go through the homologation process to become active. Let α be an artifact which goes through the Homologation Process and $\mathcal{A}$ be the group of actors who created it. Let $\mathcal{B}$ be a group of actors disjoint from $\mathcal{A}$, which performs the homologation process on α. The group $\mathcal{B}$ evaluates the artifact α to determine whether it is added to the Active System. There are two possible outcomes: α is homologated becoming part of the Active System, or it is rejected remaining in the Inactive System.

9.1.1 Algorithm

Let α be an artifact which goes through the homologation process and $\mathcal{A} = \{a_1, a_2, \ldots, a_n\}$, where $n \geq 1$, be the group of actors who created it. Let $\mathcal{B} = \{b_1, b_2, \ldots, b_q\}$, where $q \geq 1$, be the group of actors which performs the Homologation Process on α.

Algorithm 1 Homologation Process - Part 1/2

```
1: procedure HomologationProcess(Artifact α, Group of Actors 𝓑)
2:   ACTION: α is evaluated for homologation by the group 𝓑.[1]
3:   if α.homologated == False then                                  ▷ The Artifact is rejected.
4:     for each b[i] ∈ 𝓑 do[2]
5:       α.RejectedBy[i] ← b[i]                                      ▷ Identifies who rejected the Artifact.
6:       b[i].Rejected ← α[3]                                        ▷ Adds α to the list of Artifacts rejected by b[i].
7:     end for
8:     CreationProcess(Group of Actors 𝓑, New Artifact RHR 𝓡)   ▷ 𝓑 creates a new Rationale for Homologation or
   Rejection (RHR).
9:     α.RejectionJustifiedBy ← 𝓡                                  ▷ Identifies the Rationale which justifies the rejection of α.
10:    𝓡.JustifiesRejectionOf ← α
11:    Return NULL
12:  end if
13:  if α.homologated == True then                                   ▷ The Artifact is homologated.
14:    for each b[i] ∈ 𝓑 do
15:      α.HomologatedBy[i] ← b[i]                                   ▷ Identifies who homologated the Artifact.
16:      b[i].Homologated ← α                                        ▷ Adds α to the list of Artifacts homologated by b[i].
17:    end for
18:    CreationProcess(Group of Actors 𝓑, New Artifact RHR 𝓡)   ▷ 𝓑 creates a new Rationale for Homologation or
   Rejection.
19:    α.HomologationJustifiedBy ← 𝓡                               ▷ Identifies the Rationale which justifies the homologation of α.
20:    𝓡.JustifiesHomologationOf ← α
21:    ▷ It may be necessary to generate more traceability links for specific situations, as follows.
22:    if α.newlyCreated == True AND α.hasReifiedFromLinks == True then   ▷ Newly created Artifact and has
   ReifiedFrom links starting from it.
23:      for each α.ReifiedFrom[i] do                                ▷ For each Artifact α is reified from.
24:        α.ReifiedFrom[i].ReifiedTo ← α                            ▷ Create a ReifiedTo link type having α as its destination.
25:      end for
26:    end if
27:    switch α.Type do
28:      case RM
29:        α.DefinesModificationOf.ModificationDefinedBy ← α ▷ Identifies α as a Rationale having β as its target.
30:      case RR
31:        α.DefinesRemovalOf.RemovalDefinedBy ← α
32:      case RD
33:        α.DefinesDecompositionOf.DecompositionDefinedBy ← α
34:      case RAc
35:        α.DefinesActivationOf.ActivationDefinedBy ← α
36:      case RAp
37:        α.JustifiesApplicationOf.ApplicationJustifiedBy ← α
38:    end switch
39:    if α.hasDecomposedFromLinks == True then                     ▷ α is a decomposition of another Artifact.
40:      α.DecomposedFrom.DecomposedTo ← α
41:    end if
42:    if α.newlyCreated == True AND α.hasAppliesToLinks == True then ▷ Newly created Artifact which should
   be applied to other artifacts.
43:      for each α.AppliesTo[i] do                                  ▷ For each Artifact α applies to.
44:        α.AppliesTo[i].SubjectToApplicationOf ← α
45:      end for
46:    end if
47:    if α.newlyCreated == True AND α.hasDependsOnLinks == True then
48:      for each α.DependsOn[i] do                                  ▷ For each Artifact α depends on.
49:        α.DependsOn[i].NecessaryFor ← α
50:      end for
51:    end if
52:    if α.newlyCreated == True AND α.hasDependsOnPartialLinks == True then
53:      for each α.DependsOnPartial[i] do
54:        α.DependsOnPartial[i].NecessaryForPartial ← α
55:      end for
56:    end if
```

Algorithm 1 Homologation Process - Part 2/2

```
57:         if α.newlyCreated == True AND α.hasNecessaryForLinks == True then
58:             for each α.NecessaryFor[i] do
59:                 α.NecessaryFor[i].DependsOn ← α
60:             end for
61:         end if
62:         if α.newlyCreated == True AND α.hasNecessaryForPartialLinks == True then
63:             for each α.NecessaryForPartial[i] do
64:                 α.NecessaryForPartial[i].DependsOn ← α
65:             end for
66:         end if
67:         if α.newlyCreated == False AND α.NumberOfUnfulfilledDependencies > 0 then
68:             ListOfNewDependencies[] ← GetsListOfNewDependencies(α)   ▷ Dependencies which are solvable by active
    artifacts.
69:             for each ListOfNewDependencies[i] do
70:                 α.DependsOn = ListOfNewDependencies[i]
71:                 ListOfNewDependencies[i].DependsOn ← α
72:                 α.NumberOfUnfulfilledDependencies–
73:             end for
74:             if α.NumberOfUnfulfilledDependencies > 0 then   ▷ If there are dependencies which could not be solved by
    current artifacs.
75:                 ACTION: Create, modify, or activate Artifacts to fulfill the dependencies.
76:             end if
77:         end if
78:         if α.newlyCreated == True AND α.hasConflictsWithLinks == True then
79:             ACTION: Modify or remove the conflicting artifacts from the Active System to solve the conflicts.
80:         end if
81:         if α.newlyCreated == False AND (ListOfNewConflicts ← ReturnsListOfNewConflicts(α)^4) IS NOT NULL
    then ▷ The Artifact is a new version of a modified Artifact and its modification resulted in the generation of conflicts.
82:             for each ListOfNewConflicts[i] do
83:                 α.ConflictsWith ← ListOfNewConflicts[i]
84:                 ListOfNewConflicts[i].ConflictsWith ← α
85:             end for
86:             ACTION: Modify or remove the conflicting artifacts from the Active System to solve the conflicts.
87:         end if
88:         if α.newlyCreated == False AND (ListOfSolvedConflicts ← ReturnsListOfSolvedConflicts(α)) IS NOT NULL
    then
89:             for each ListOfSolvedConflicts[i] do
90:                 RemoveArtifactFromList(ListOfSolvedConflicts[i].ConflictsWith, α)^5
91:                 RemoveArtifactFromList(α.ConflictsWith, ListOfSolvedConflicts[i])
92:             end for
93:         end if
94:         Activate(α)                                         ▷ α is added to the Active System.
95:         Return Success
96:     end if
97: end procedure
```

9.1.2 Textual Description

Every newly created artifact starts off inactive; it has to go through the Homologation Process to become active. Let α be an artifact which goes through the homologation process and $\mathcal{A} = \{a_1, a_2, \ldots, a_n\}$, where $n \geq 1$, be the group of actors who created it. Let

[1] Statements starting with ACTION define actions to be performed which are beyond the scope of the process description; e.g., the actual action of modifying a code fragment.

[2] The loop For increases its index by one automatically.

[3] To simplify, we define: artifact1.list ← artifact2 means artifact2 is assigned to the last index of artifact1.list; artifact1.list[predefinedIndex] ← artifact2 means artifact2 is assigned to predefinedIndex of artifact1.list.

[4] Returns the new conflicts resulting from the modification of α.

[5] A function which, given a lists of elements and an Artifact, removes the Artifact from the list.

$\mathcal{B} = \{b_1, b_2, \ldots, b_q\}$, where $q \geq 1$, be a group of actors disjoint from $\mathcal{A}$, i.e., $\mathcal{A} \cap \mathcal{B} = \emptyset$, which performs the homologation process on α. The group $\mathcal{B}$ evaluates the artifact α to determine whether it is added to the Active System. There are two possible outcomes: α is homologated becoming part of the Active System, or it is rejected remaining in the Inactive System.

If α is homologated, two sets of links are generated: $E = \{e_1, e_2, \ldots, e_q\}$ and $F = \{f_1, f_2, \ldots, f_q\}$. Each $e_i \in E$, $1 \leq i \leq q$, is an *HomologatedBy* link starting in α and having $b_i \in \mathcal{B}$ as its destination. Each $f_i \in F$, $1 \leq i \leq q$, is an *Homologated* link starting in $b_i \in \mathcal{B}$ and having α as its destination. Then, the group of actors $\mathcal{B}$ creates an RHR artifact $\mathcal{R}$ – which will not be activated – and two links are generated: e_α and e_R. Link e_α is an *HomologationJustifiedBy* link starting in α and having $\mathcal{R}$ as its destination. Link e_R is a *JustifiesHomologationOf* link starting in $\mathcal{R}$ and having α as its destination. The artifact $\mathcal{R}$ does not undergo the Creation Process but its creators must be identified; therefore, two sets of links are generated: $U = \{u_1, u_2, \ldots, u_q\}$ and $W = \{w_1, w_2, \ldots, w_q\}$. Each $u_i \in U$, $1 \leq i \leq q$, is a *CreatedBy* link starting in $\mathcal{R}$ and having $b_i \in \mathcal{B}$ as its destination. Each $w_i \in W$, $1 \leq i \leq q$, is a *Created* link starting in $b_i \in \mathcal{B}$ and having $\mathcal{R}$ as its destination. The artifact $\mathcal{R}$ justifies the homologation of α.

If α is rejected, two sets of links are generated: $E = \{e_1, e_2, \ldots, e_q\}$ and $F = \{f_1, f_2, \ldots, f_q\}$. Each $e_i \in E$, $1 \leq i \leq q$, is a *RejectedBy* link starting in α and having $b_i \in \mathcal{B}$ as its destination. Each $f_i \in F$, $1 \leq i \leq q$, is a *Rejected* link starting in $b_i \in \mathcal{B}$ and having α as its destination. Then, the group of actors $\mathcal{B}$ creates an RHR artifact $\mathcal{R}$ – which will not be activated – and two links are generated: e_α and e_R. Link e_α is a *RejectionJustifiedBy* link starting in α and having $\mathcal{R}$ as its destination. Link e_R is a *JustifiesRejectionOf* link starting in $\mathcal{R}$ and having α as its destination. The artifact $\mathcal{R}$ does not undergo the creation process but its creators must be identified; therefore, two sets of links are generated: $U = \{u_1, u_2, \ldots, u_q\}$ and $W = \{w_1, w_2, \ldots, w_q\}$. Each $u_i \in U$, $1 \leq i \leq q$, is a *CreatedBy* link starting in $\mathcal{R}$ and having $b_i \in \mathcal{B}$ as its destination. Each $w_i \in W$, $1 \leq i \leq q$, is a *Created* link starting in $b_i \in \mathcal{B}$ and having $\mathcal{R}$ as its destination. The artifact $\mathcal{R}$ justifies the rejection of α.

It may be necessary to generate more traceability links for specific situations, as follows.

If α is homologated, is a newly created artifact, and a reification of other artifacts, having *ReifiedFrom* links starting from it, it will be necessary to generate opposite links. Let $\beta = \{\beta_1, \beta_2, \ldots, \beta_m\}$ be the set of m artifacts reified into α; a set of links $E = \{e_1, e_2, \ldots, e_m\}$ is generated, where each $e_i \in E$, $1 \leq i \leq m$, is a *ReifiedTo* link starting in $\beta_i \in \beta$ and having α as its destination.

If α is homologated and is a rationale type, having a *DefinesModificationOf*, *DefinesDecompositionOf*, *DefinesRemovalOf*, *DefinesActivationOf*, or *JustifiesApplicationOf* link starting from it, it will be necessary to generate an opposite link. Let β be the target of α. If α is an RM, a *ModificationDefinedBy* link is generated starting in β and having α as its destination. If α is an RD, a *DecompositionDefinedBy* link is generated starting in β and having α as its destination. If α is an RR, a *RemovalDefinedBy* link is generated starting in β and having α as its destination. If α is an RAc, an *ActivationDe-*

finedBy link is generated starting in β and having α as its destination. If α is an RAp, an *ApplicationJustifiedBy* link is generated starting in β and having α as its destination.

If α is homologated and a decomposition of another artifact, having a *DecomposedFrom* link starting from it, it will be necessary to generate an opposite link. Let β be the parent of α. A *DecomposedTo* link is generated starting in β and having α as its destination.

If α is homologated, is not a newly created artifact (it is a new version of a modified artifact), having a *ModifiedFrom* link starting from it, it will be necessary to generate an opposite link. Let β be the previous version of α. A *ModifiedTo* link is generated starting in β and having α as its destination.

If α is homologated, is a newly created artifact, and should be applied to other artifacts, having *AppliesTo* links starting from it, it will be necessary to generate opposite links. Let $\beta = \{\beta_1, \beta_2, \ldots, \beta_m\}$ be the set of m target artifacts of α; a set of links $E = \{e_1, e_2, \ldots, e_m\}$ is generated, where each $e_i \in E$, $1 \leq i \leq m$, is a *SubjectToApplicationOf* link starting in $\beta_i \in \beta$ and having α as its destination.

If α is homologated, is a newly created artifact, and depends on other artifacts, having *DependsOn* links starting from it, it will be necessary to generate opposite links. Let $\beta = \{\beta_1, \beta_2, \ldots, \beta_m\}$ be the set of m artifacts necessary for α; a set of links $E = \{e_1, e_2, \ldots, e_m\}$ is generated, where each $e_i \in E$, $1 \leq i \leq m$, is a *NecessaryFor* link starting in $\beta_i \in \beta$ and having α as its destination. If there are other artifacts satisfying $\beta_i \in \beta$, given a certain dependency, a $NecessaryFor^{Partial}$ link must be created instead of a *NecessaryFor* link.

If α is homologated, is a newly created artifact, and is necessary for other artifacts, having *NecessaryFor* links starting from it, it will be necessary to generate opposite links. Let $\beta = \{\beta_1, \beta_2, \ldots, \beta_m\}$ be the set of m artifacts dependent on α; a set of links $E = \{e_1, e_2, \ldots, e_m\}$ is generated, where each $e_i \in E$, $1 \leq i \leq m$, is a *DependsOn* link starting in $\beta_i \in \beta$ and having α as its destination. If there is more than one artifact satisfying α, given a certain dependency, a $DependsOn^{Partial}$ link must be created instead of a *DependsOn* link.

If α is homologated, is not a newly created artifact (it is a new version of a modified artifact), and has new dependencies on other artifacts, it will be necessary to generate links. Let $\beta = \{\beta_1, \beta_2, \ldots, \beta_m\}$ be the set of m artifacts necessary for α. Two sets of links are generated: $E = \{e_1, e_2, \ldots, e_m\}$ and $F = \{f_1, f_2, \ldots, f_m\}$. Each $e_i \in E$, $1 \leq i \leq m$, is a *DependsOn* link starting in α and having $\beta_i \in \beta$ as its destination. Each $f_i \in F$ is a *NecessaryFor* link starting in $\beta_i \in \beta$ and having α as its destination.

The artifact α, after being homologated, may have dependencies which are not fulfilled by any artifact in the Active System. These dependencies may be fulfilled through the creation, modification, or activation of artifacts.

If α is homologated, is a newly created artifact, and has conflicts with other artifacts in the Active System, having *ConflictsWith* links starting from it, it will be necessary to remove the destination artifacts or to modify them to solve the conflicts.

If α is homologated, is a new version of a modified artifact, and its modification resulted in the generation of conflicts with other artifacts, it will be necessary to generate the corresponding links. Let $\beta = \{\beta_1, \beta_2, \ldots, \beta_m\}$ be the set of m artifacts in conflict with α. Two sets of links are generated: $E = \{e_1, e_2, \ldots, e_m\}$ and $F = \{f_1, f_2, \ldots, f_m\}$. Each $e_i \in E$, $1 \leq i \leq m$, is a *ConflictsWith* link starting in α and having $\beta_i \in \beta$ as

its destination. Each $f_i \in F$ is a *ConflictsWith* link starting in $\beta_i \in \beta$ and having α as its destination. If any of the conflicting artifacts are in the Active System, it will be necessary to remove or to modify them to solve the conflicts.

If α is homologated, is a new version of a modified artifact, and its modification solved previous conflicts with other artifacts, it will be necessary to delete the corresponding links.

9.1.3 Assessing the Impact of Homologating a Rationale for Modification or a Rationale for Removal

The decision to homologate a Rationale for Modification (RM) or a Rationale for Removal (RR) is contingent upon the impact in the Active System; it may be necessary to perform a sequence of actions to deal with the problems arisen from the actions of modification or removal. Hence, the homologation may not occur if the impact is undesirable. This process provides the necessary steps to be taken given each particular situation, enabling the user to assess the impact, and consequently the cost, of homologating a Rationale for Modification or a Rationale for Removal in a project.

9.1.3.1 Algorithm

Let α be an RM or RR, and β be the target of α. Let $G = \{g_1, g_2, \ldots, g_r\}$, where $r \geq 1$, be the set of all links starting in β and $\mathcal{C} = \{c_1, c_2, \ldots, c_r\}$ be the set of destination artifacts of the links in G. Let $H = \{h_1, h_2, \ldots, h_r\}$ be the set of opposite links to G, having β as its destination, where g_i, $1 \leq i \leq r$, is opposite to h_i. Each artifact $c_i \in \mathcal{C}$ must be examined to decide whether α should be homologated.

Algorithm 2 Assessing the Impact of Homologating a Rationale for Modification or a Rationale for Removal - Part 1/4

```
1: procedure AssessmentRMAndRR-HomologationProcess(Artifact α, Group of Actors B)
2:     β ← α.RationaleTarget
3:     if α.type == RM then
4:         for each β.NecessaryFor[i] do
5:             if β.NecessaryFor[i] will not be able to depend on β after the modification then
6:                 ACTION: Three alternatives: (i) modify β.NecessaryFor[i] to end its dependency to β, (ii) deactivate
   β.NecessaryFor[i], or (iii) create, modify, or activate artifacts to fulfill β.NecessaryFor[i] dependency.
7:                 if It was decided to solve the issue without deactivating β.NecessaryFor[i] then
8:                     DeleteLink(g[i])
9:                     DeleteLink(h[i])
10:                end if
11:            end if
12:        end for
13:        for each β.NecessaryForPartial[i] do
14:            if β.NecessaryForPartial[i] will not be able to depend on β after the modification then
15:                DeleteLink(g[i])
16:                DeleteLink(h[i])
17:            end if
18:        end for
```

Algorithm 2 Assessing the Impact of Homologating a Rationale for Modification or a Rationale for Removal - Part 2/4

19: **for each** β.DependsOn[i] **do**
20: **if** β.DependsOn[i] will not be necessary for β after the modification **then**
21: DeleteLink($g[i]$)
22: DeleteLink($h[i]$)
23: *ACTION: β.DependsOn[i] should be removed if it only exists to fulfill the dependency of β.*
24: **end if**
25: **end for**
26: **for each** β.DependsOnPartial[i] **do**
27: **if** β.DependsOnPartial[i] will not be necessary for β after the modification **then**
28: DeleteLink($g[i]$)
29: DeleteLink($h[i]$)
30: *ACTION: β.DependsOnPartial[i] should be removed if it only exists to fulfill the dependency of β.*
31: **end if**
32: **end for**
33: **for each** β.ReifiedFrom[i] **do**
34: **if** β will not have information from β.ReifiedFrom[i] after the modification **then**
35: *ACTION: β.ReifiedFrom[i] should be removed if it only propagates information to β. If not, it should be modified to remove the related information.*
36: **end if**
37: **end for**
38: **for each** β.ReifiedTo[i] **do**
39: **if** The modification will remove information corresponding to all of β.ReifiedTo[i] **then**
40: *ACTION: β.ReifiedFrom[i] should be removed.*
41: **else if** The modification will remove information corresponding to part of β.ReifiedTo[i] **then**
42: *ACTION: The corresponding information should be removed by modifying β.ReifiedTo[i].*
43: **else if** The modification will add new information to β **then**
44: *ACTION: The corresponding information should be propagated to other artifacts, by creation or modification.*
45: **end if**
46: **end for**
47: **for each** β.AppliesTo[i] **do**
48: **if** β can not apply to β.AppliesTo[i] after the modification. **then**
49: **if** It is not necessary that β, or another artifact, applies to β.AppliesTo[i] **then**
50: DeleteLink($g[i]$)
51: DeleteLink($h[i]$)
52: **else if** It is necessary that β applies to β.AppliesTo[i] **then**
53: *ACTION: β.AppliesTo[i] should be modified accordingly.*
54: DeleteLink($g[i]$)
55: DeleteLink($h[i]$) ▷ These links should be regenerated later, after the modification.
56: **else if** It is necessary that another artifact applies to β.AppliesTo[i] **then**
57: *ACTION: Such an artifact must be created, modified, or activated.*
58: DeleteLink($g[i]$)
59: DeleteLink($h[i]$)
60: **end if**
61: **end if**
62: **end for**
63: **for each** β.SubjectToApplicationOf[i] **do**
64: **if** β can not be subject to the application of β.SubjectToApplicationOf[i] after the modification **then**
65: **if** It is not necessary that β is subjected to the application of β.SubjectToApplicationOf[i] **then**
66: DeleteLink($g[i]$)
67: DeleteLink($h[i]$)
68: **else if** It is necessary that β is subjected to the application of β.SubjectToApplicationOf[i] **then**
69: *ACTION: β.SubjectToApplicationOf[i] should be modified accordingly.*
70: DeleteLink($g[i]$)
71: DeleteLink($h[i]$) ▷ These links should be regenerated later, after the modification.
72: **else if** It is necessary that β is subjected to the application of other artifact **then**
73: *ACTION: Such artifact must be created, modified, or activated.*
74: DeleteLink($g[i]$)
75: DeleteLink($h[i]$)
76: **end if**
77: **end if**
78: **end for**

Algorithm 2 Assessing the Impact of Homologating a Rationale for Modification or a Rationale for Removal - Part 3/4

```
79:       for each β.ComprisedOf[i] do
80:           ACTION: reclassify parts of β as needed.
81:       end for
82:       if New dependencies will be generated after the modification then
83:           if There are dependencies which can not be fulfilled by Artifacts from the Active System then
84:               ACTION: Create, modify, or activate Artifacts to fulfill the dependencies.
85:           end if
86:       end if
87:       if Dependencies will be solved after the modification then
88:           if There are Artifacts which exist only to fulfill these dependencies then
89:               ACTION: Remove the corresponding Artifacts from the Active System.
90:           end if
91:       end if
92:       if New conflicts will be generated after the modification then
93:           ACTION: Modify or remove the corresponding Artifacts to solve the conflicts.
94:       end if
95:       if Conflicts will be solved after the modification then
96:           ACTION: Delete the corresponding links.
97:       end if
98:   end if
99:   if α.type == RR then
100:      for each β.NecessaryFor[i] do
101:          ACTION: Three alternatives: (i) modify β.NecessaryFor[i] to end its dependency to β, (ii) deactivate
      β.NecessaryFor[i], or (iii) create, modify, or activate artifacts to fulfill β.NecessaryFor[i] dependency.
102:      end for
103:      for each β.DependsOn[i] do
104:          if β.DependsOn[i] exists only to fulfill a dependency from β then
105:              ACTION: β.DependsOn[i] should be removed from the Active System
106:          end if
107:      end for
108:      for each β.DependsOnPartial[i] do
109:          if β.DependsOnPartial[i] exists only to fulfill a dependency from β then
110:              ACTION: β.DependsOnPartial[i] should be removed from the Active System
111:          end if
112:      end for
113:      for each β.ReifiedFrom[i] do
114:          if The information propagated to β should be kept in the Active System then
115:              DeleteLink(g[i])
116:              DeleteLink(h[i])
117:              ACTION: Create, modify, or activate Artifacts to propagate the information.
118:          else
119:              ACTION: Modify β.ReifiedFrom[i] to remove the information. If it corresponds to the entirety of
      β.ReifiedFrom[i], β.ReifiedFrom[i] should be removed instead.
120:          end if
121:      end for
122:      for each β.ReifiedTo[i] do
123:          ACTION: Modify β.ReifiedTo[i] to remove the information propagated from β. If it corresponds to the
      entirety of β.ReifiedTo[i], β.ReifiedTo[i] should be removed instead.
124:      end for
125:      for each β.AppliesTo[i] do
126:          if It is not necessary that another artifact applies to β.AppliesTo[i] then
127:              DeleteLink(g[i])
128:              DeleteLink(h[i])
129:          else
130:              ACTION: Such an artifact must be created, modified, or activated.
131:              DeleteLink(g[i])
132:              DeleteLink(h[i])
133:          end if
134:      end for
```

Algorithm 2 Assessing the Impact of Homologating a Rationale for Modification or a Rationale for Removal - Part 4/4

```
135:        for each β.SubjectToApplicationOf[i] do
136:            if It is not necessary that another artifact is subjected to the application of β.SubjectToApplicationOf[i]
    then
137:                DeleteLink(g[i])
138:                DeleteLink(h[i])
139:            else
140:                ACTION: Such artifact must be created, modified, or activated.
141:                DeleteLink(g[i])
142:                DeleteLink(h[i])
143:            end if
144:        end for
145:    end if
146:    if There are necessary modifications on associated artifacts then    ▷ It may be necessary to modify artifacts
    related to β; e.g., removing function calls in other artifacts.
147:        ACTION: Perform, or concatenate, modifications on the associated artifacts.
148:    end if
149: end procedure
```

9.1.3.2 Textual Description

Let α be an RM or RR and β be the target of α. Let $G = \{g_1, g_2, \ldots, g_r\}$ be the set of all links starting in β and $\mathcal{C} = \{c_1, c_2, \ldots, c_r\}$ be the set of destination artifacts of the links in G. Let $H = \{h_1, h_2, \ldots, h_r\}$ be the set of opposite links to G, having β as its destination, where g_i is opposite to h_i. If $\exists g_i \in G$, $1 \leq i \leq r$, g_i being a *NecessaryFor*, *DependsOn*, *NecessaryFor*^{Partial}, *DependsOn*^{Partial}, *ConflictsWith*, *ReifiedFrom*, *ReifiedTo*, *AppliesTo*, *SubjectToApplicationOf*, *PartOf*, or *ComprisedOf* link, each artifact $c_i \in \mathcal{C}$ must be examined to decide whether α should be homologated.

If α is an RM and it is going to be homologated, for each *NecessaryFor* link g_i, unfulfilled dependencies resulting from the homologation of α must be dealt with. I.e., is there any c_i which can not depend on β after the modification defined by α? If there is, each c_i having an unfulfilled dependency: (i) must be modified to end its dependency to β; (ii) be removed from the Active System; or (iii) artifacts must be created, modified, or activated to fulfill the dependencies. If c_i is not removed, g_i and h_i must be deleted, since c_i will not depend on β anymore. If c_i is removed, the dependency information between c_i and the previous version of β, now in the Inactive System, should be kept; therefore, the corresponding g_i and h_i will not be deleted, being kept for historical reasons or to be useful in a future activation.

If α is an RR and it is going to be homologated, for each *NecessaryFor* link g_i, there is going to be an unfulfilled dependency resulting from the homologation of α. Artifact c_i still depends on β, but since β is being moved to the Inactive System, it can not satisfy c_i. To solve this: (i) c_i must be modified to end its dependency to β; (ii) c_i be removed from the Active System; or (iii) artifacts must be created, modified, or activated to fulfill the dependency of c_i.

If α is an RM and it is going to be homologated, for each *NecessaryFor*^{Partial} link g_i, if there is any c_i which can not depend on β after the modification defined by α, the links g_i and h_i must be deleted. It is not necessary to perform more actions since there is at least one other artifact fulfilling the same dependency.

If α is an RR and it is going to be homologated, for each *NecessaryFor*^{Partial} link g_i, it is not necessary to perform more actions since there is at least one other artifact fulfilling the same dependency.

If α is an RM and it is going to be homologated, for each *DependsOn* or *DependsOn*^{Partial} link g_i, fulfilled dependencies resulting from the homologation of α must be taken into account; i.e., is there any c_i which is not anymore necessary for β after the modification defined by α? If there is, the corresponding g_i and h_i must be deleted. If c_i exists only to fulfill a dependency of β, c_i should be removed. In this case, the dependency information between c_i and the previous version of β, now in the Inactive System, should be kept; therefore, the corresponding g_i and h_i will not be deleted.

If α is an RR and it is going to be homologated, for each *DependsOn* or *DependsOn*^{Partial} link g_i, for each c_i which exists only to fulfill a dependency of β[1], c_i should be removed. The g_i and h_i links between c_i and β will not be deleted.

If α is an RM and it is going to be homologated, for each *ReifiedFrom* link g_i where β, after its modification, will not have any information from the corresponding c_i, c_i will have to be modified or removed. The former happens when the modification removes part of the information from c_i; i.e., c_i propagates information to other artifacts in the Active System. The links g_i and h_i will remain connecting the previous versions of c_i and β, but there will not be g_i and h_i links connecting the new versions of c_i and β. The latter happens when the modification removes, from β, all the information from c_i, and c_i only propagates information to β. The links g_i and h_i will remain connecting c_i and the previous version of β, both of them in the Inactive System. No unfulfilled dependencies will arise from the removal of c_i since β is a reification from c_i. If c_i is necessary for an artifact, β is necessary too; removal of information from β means that dependencies have been, or will be, fulfilled.

If α is an RR and it is going to be homologated, for each *ReifiedFrom* link g_i, it must be evaluated whether the information propagated from c_i to β should be kept in the Active System. (i) If that is not the case, c_i should be modified to remove the corresponding information propagated to β; the links g_i and h_i will remain connecting the previous version of c_i and β, both of them in the Inactive System. If the information in β corresponds to all the information from c_i, c_i should be removed too. The links g_i and h_i will remain connecting c_i and β, both of them in the Inactive System. (ii) If the information should be kept, the links g_i and h_i should be removed and the information should be added by: the creation of new artifacts; the modification of current artifacts; or the activation of inactive artifacts. New *ReifiedFrom* and *ReifiedTo* links should be generated to connect c_i to these artifacts.

If α is an RM and it is going to be homologated, for each *ReifiedTo* link g_i where c_i is affected by the modification of β: (i) if the modification removes information corresponding to all of c_i, c_i should be removed. The links g_i and h_i will remain connecting c_i and the previous version of β, both of them in the Inactive System. (ii) If the modification removes information corresponding to part of c_i, c_i should be modified accordingly; i.e., this information should be removed from c_i. The links g_i and h_i will remain connecting

[1] This should not occur with *DependsOn*^{Partial} links; however, this is a possible situation.

the previous versions of c_i and β, but there will not be g_i and h_i links connecting the new versions of c_i and β. (iii) If the modification adds new information to β, it should be propagated to other artifacts; these can be new or current artifacts.

If α is an RR and it is going to be homologated, for each *ReifiedTo* link g_i, c_i should be modified to remove the information from β. If the information from β corresponds to all of c_i, c_i should be removed. The links g_i and h_i will remain connecting c_i and β, both of them in the Inactive System. If the information from β corresponds to part of c_i, c_i should be modified accordingly; i.e., this information should be removed from c_i. The links g_i and h_i will remain connecting the previous versions of c_i and β, both of them in the Inactive System.

If α is an RM and it is going to be homologated, for each *AppliesTo* link g_i where β does not apply to c_i anymore: (i) if it is not necessary that β, or another artifact, applies to c_i, links g_i and h_i should be deleted; (ii) if it is necessary that β applies to c_i, the artifact c_i should be modified accordingly, and links g_i and h_i should be deleted to be regenerated later; (iii) if it is necessary that another artifact applies to c_i, such an artifact must be created, modified, or activated, and links g_i and h_i should be deleted.

If α is an RR and it is going to be homologated, for each *AppliesTo* link g_i: (i) if it is not necessary that another artifact applies to c_i, links g_i and h_i should be deleted; (ii) if it is necessary that another artifact applies to c_i, such an artifact must be created, modified, or activated, and links g_i and h_i should be deleted.

If α is an RM and it is going to be homologated, for each *SubjectToApplicationOf* link g_i where β is not subject to the application of c_i after the modification: (i) if it is not necessary that β is subjected to the application of c_i, or another artifact, links g_i and h_i should be deleted; (ii) if it is necessary that β is subjected to the application of c_i, the artifact c_i should be modified accordingly, and links g_i and h_i should be deleted to be regenerated later; (iii) if it is necessary that β is subjected to the application of some artifact, such an artifact must be created, modified, or activated, and links g_i and h_i should be deleted.

If α is an RR and it is going to be homologated, for each *SubjectToApplicationOf* link g_i: (i) if it is not necessary that another artifact is subjected to the application of c_i, links g_i and h_i should be deleted; (ii) if it is necessary that another artifact is subjected to the application of c_i, such an artifact must be created, modified, or activated, and links g_i and h_i should be deleted.

If α is an RM and it is going to be homologated, if there are *ComprisedOf* links starting in β, β should have its parts reclassified accordingly; it may be necessary to classify new parts or remove existing parts, adding or deleting *PartOf* and *ComprisedOf* links.

If α is an RR and it is going to be homologated, for each *ComprisedOf* link g_i, no action is necessary since each c_i is part of β.

If α is an RM and it is going to be homologated, for each *PartOf* link g_i, no action is necessary since the modification of β has no effect in the relationship with the parent artifact.

If α is an RM and it is going to be homologated, it should be assessed if dependencies are generated or solved by the modification on β described in α. (i) If dependencies are generated and there are dependencies which can not be fulfilled by artifacts in the

Active System, it will be necessary to modify, activate, or create artifacts to solve these dependencies. (ii) If dependencies are solved by the modifications described in α and there are artifacts which exist only to fulfill β dependencies, these should be removed from the Active System.

If α is an RM and it is going to be homologated, it should be assessed if conflicts are generated or solved by the modification on β described in α. (i) If conflicts are generated, it will be necessary to modify or remove the conflicting artifacts to solve the conflicts. (ii) If conflicts are solved, nothing should be done besides deleting the corresponding links.

If α is an RR and it is going to be homologated, it should be assessed if there are necessary modifications on other artifacts as the result of removing β (see Section 9.5); if so, it will be necessary to modify the corresponding artifacts after the removal process.

9.1.4 Why is the Homologation Process Useful?

Homologation avoids the introduction of inconsistencies in the Active System; e.g., the removal of a necessary artifact could occur without the evaluation prescribed by the homologation process, leaving the system inconsistent. Furthermore, it enables the evaluation of an action before it is performed. Therefore, the decision whether to proceed with an action may be guided through the assessment of its impact; e.g., during the homologation process it may be found that the cost of the removal of an artifact outweighs its benefits.

9.2 Modification Process

Let α be an artifact which goes through the Modification Process. A group of actors $\mathcal{A}$ creates an Rationale for Modification P having α as its target. The artifact P should be homologated for the Modification process to take place. Following the homologation of P, an identical copy of α (α^1) is created. An identical copy of an artifact has the same contents and structure; the links originating in the artifact are also copied, having the same destination artifacts, but the links having α as their destination will not be copied. It is possible to trace from the new version to the previous version and vice-versa.

A group of actors $\mathcal{B}$ performs the modification described in P on α^1 (the group $\mathcal{B}$ is not necessarily disjoint from $\mathcal{A}$). The artifact α^1 was created and modified but it is still in the Inactive System; it must be homologated to become part of the Active System. The modification must be evaluated and validated, i.e. homologated, according to what was proposed in P, by a group of actors disjoint of $\mathcal{B}$. The successful homologation of α^1 results in moving it from the Inactive System to the Active System, and moving α from the Active System to the Inactive System. Finally, P is moved to the Inactive System since it has fulfilled its role.

9.2.1 Algorithm

Let α be an artifact which goes through the Modification Process. A group of actors $\mathcal{A}$ $= \{a_1, a_2, \ldots, a_n\}$, where $n \geq 1$, creates an Rationale for Modification P having α as its

target. A group of actors $\mathcal{D} = \{d_1, d_2, \ldots, d_l\}$, where $l \geq 1$, homologates artifact P for the Modification process to take place. Following its homologation, a group actors $\mathcal{B} = \{b_1, b_2, \ldots, b_m\}$, where $m \geq 1$, performs the modification described in P.

Algorithm 3 Modification Process

```
1: procedure ModificationProcess(Artifact α, Group of Actors ℬ, Group of Actors 𝒟, Rationale for Modification P)
2:     α¹ ← CreatesCopy(α)                        ▷ Creates an identical copy of the Artifact.
3:     α¹.ModifiedFrom ← α                        ▷ Indicates its previous version.
4:     ACTION: The group ℬ performs the modification described in P to α¹.
5:     for each b[i] ∈ ℬ do
6:         α¹.ModifiedBy[i] ← b[i]                ▷ Identifies who modified the Artifact.
7:         b[i].Modified ← α¹                     ▷ Adds α¹ to the list of Artifacts modified by b[i].
8:     end for
9:     HomologationProcess(α¹, 𝒟)                 ▷ The group of actors 𝒟 will consider the homologation of α¹.
10:    if α¹.homologated == True then
11:        Deactivate(P)                          ▷ α¹ was added to the Active System and P is moved to the Inactive System.
12:        RedirectLinks(α,α¹)                    ▷ Redirect the links arriving in α to α¹.
13:        Return α¹
14:    else
15:        Return NULL                            ▷ The homologation failed.
16:    end if
17: end procedure
```

9.2.2 Textual Description

The modification of an artifact is the process of copying it and performing the modification on this copy. This process is composed of two steps: generation of new links and rearrangement of existing links.

Generating Links

Let α be an artifact which goes through the Modification Process. A group of n actors $\mathcal{A} = \{a_1, a_2, \ldots, a_n\}$, where $n \geq 1$, creates an Rationale for Modification P having α as its target. The artifact P should be homologated for the Modification process to take place. Following the homologation of P, α^1 – an identical copy of α – is created. An identical copy of an artifact has the same contents and structure; the links originating in the artifact are also copied, having the same destination artifacts, but the links having α as their destination will not be copied. A new *ModifiedFrom* link is generated, starting in α^1 and having α as its destination; i.e., the new version of the artifact indicates its previous version. A group of q actors $\mathcal{B} = \{b_1, b_2, \ldots, b_q\}$, where $q \geq 1$, performs the modification described in P on α^1 (the group $\mathcal{B}$ is not necessarily disjoint from $\mathcal{A}$). As a result, two sets of links are generated: $E = \{e_1, e_2, \ldots, e_q\}$ and $F = \{f_1, f_2, \ldots, f_q\}$. Each $e_i \in E$ is a *ModifiedBy* link starting in α^1 and having $b_i \in \mathcal{B}$ as its destination. Each $f_i \in F$ is a *Modified* link starting in $b_i \in \mathcal{B}$ and having α^1 as its destination. The artifact α^1 was created and modified but it is still in the Inactive System; it must be homologated to become part of the Active System. The modification must be evaluated and validated, i.e. homologated, according to what was proposed in P, by a group of actors disjoint of $\mathcal{B}$.

The successful homologation of α^1 results in moving it from the Inactive System to the Active System, and moving α from the Active System to the Inactive System. Finally, P is moved to the Inactive System since it has fulfilled its role.

Rearranging Links

Rearrangement of links may be necessary after the replacement of α by α^1. Some of the links having α as their destination should be rearranged to have α^1 as their destination.

Definition 9.2.1. Let α, β and γ be distinct artifacts and let e be a link starting in α and having β as its destination. The redirection of e to γ is the generation of a new link e^*, of the same type and having the same information as e, starting in α and having γ as its destination. The link e is then deleted to complete the redirection.

Let $G = \{g_1, g_2, \ldots, g_r\}$ be the set of links having α as its destination and $H = \{h_1, h_2, \ldots, h_r\}$ be the set of links starting in α^1. Decisions should be made during the homologation of P regarding artifacts connected to α; these artifacts may be removed, kept unchanged or modified. For the artifacts whose modification ended the relationship, it is necessary to delete the corresponding g_i and h_i links; e.g., a dependency relationship which is not fulfilled by the current versions of the artifacts. For the artifacts whose modification did not not change the relationship, it may be necessary to redirect the related g_i.

9.3 Decomposition Process

Let α be an artifact which goes through the Decomposition Process. A group of actors $\mathcal{A}$ creates a Rationale for Decomposition P having α as its target. The artifact P should be homologated for the decomposition process to take place. Following the homologation of P, the new artifacts defined in P – the resulting artifacts of the decomposition – are created (each artifact will go through the Creation Process). Let β be the set of these artifacts. The successful homologation of each $\beta_i \in \beta$, done by a group of actors disjoint of $\mathcal{B}$, is necessary for the decomposition process to be complete. The parent artifact α and P are moved to the Inactive System following the homologation of all the artifacts in β.

9.3.1 Algorithm

A group of actors $\mathcal{A} = \{a_1, a_2, \ldots, a_n\}$, where $n \geq 1$, creates a Rationale for Decomposition P having an artifact α as its target. A group of actors $\mathcal{D} = \{d_1, d_2, \ldots, d_l\}$, where $l \geq 1$, homologates artifact P for the Decomposition Process to take place. Following its homologation, a group actors $\mathcal{B} = \{b_1, b_2, \ldots, b_q\}$, where $q \geq 1$, performs the decomposition described in P.

Let $G = \{g_1, g_2, \ldots, g_r\}$ be the set of links starting in α and let $\mathcal{C} = \{c_1, c_2, \ldots, c_r\}$ the set of destination artifacts of G. Let $H = \{h_1, h_2, \ldots, h_r\}$ be the set of opposite links to G, starting in C, where g_i is opposite to h_i.

Algorithm 4 Decomposition Process - Part 1/2

1: **procedure** DecompositionProcess(Artifact α, Group of Actors $\mathcal{B}$, Group of Actors $\mathcal{D}$, Rationale for Modification P)
2: *ACTION: $\mathcal{B}$ decomposes α into the group of Artifacts $\beta = \{\beta_1, \beta_2, \ldots, \beta_r\}$ as described in P.*
3: **for each** $b[i] \in \mathcal{B}$ **do**
4: α.DecomposedBy[i] $\leftarrow b[i]$ ▷ Identifies who decomposed the Artifact.
5: $b[i]$.Decomposed $\leftarrow \alpha$ ▷ Adds α to the list of Artifacts decomposed by $b[i]$.
6: **end for**
7: Count $\leftarrow 0$
8: **for each** $\beta[i] \in \beta$ **do**
9: HomologationProcess($\beta[i]$, $\mathcal{D}$)
10: Count++
11: **end for**
12: Deactivate(P)
13: **if** Count $<$ β.size **then** ▷ Not all $\beta[i]$ were homologated.
14: Deactivate(β) ▷ Deactivate each $\beta[i]$ successfully homologated.
15: **Return** NULL ▷ Indicates the decomposition was not successful.
16: **end if**
17: **for each** $g[i] \in G$ **do**
18: **if** $g[i]$.type $==$ *ReifiedFrom* **then**
19: **for each** $\beta[j]$ **do**
20: **if** HaveInformationFrom($\beta[j]$, $c[i]$) **then** ▷ Did $c[i]$ propagate information to α and this information was assigned to $\beta[j]$?
21: $\beta[j]$.ReifiedFrom $\leftarrow c[i]$ ▷ Adds $c[i]$ to the list of Artifacts $\beta[j]$ was reified from.
22: $c[i]$.ReifiedTo $\leftarrow \beta[j]$
23: **end if**
24: **end for**
25: DeleteLink($h[i]$) ▷ Delete the link which indicated $c[i]$ was reified into α.
26: **end if**
27: **if** $g[i]$.type $==$ *ReifiedTo* **then**
28: **for each** $\beta[j]$ **do**
29: **if** HaveInformationFrom($\beta[j]$, $c[i]$) **then**
30: $\beta[j]$.ReifiedTo $\leftarrow c[i]$
31: $c[i]$.ReifiedFrom $\leftarrow \beta[j]$
32: **end if**
33: **end for**
34: DeleteLink($h[i]$)
35: **end if**
36: **if** $g[i]$.type $==$ *DependsOn* **then**
37: **for each** $\beta[j]$ **do**
38: **if** Dependency($\beta[j]$, $c[i]$) **then**[6] ▷ If $\beta[j]$ inherited dependency to $c[i]$ from α.
39: $\beta[j]$.DependsOn $\leftarrow c[i]$
40: $c[i]$.NecessaryFor $\leftarrow \beta[j]$
41: **end if**
42: **end for**
43: DeleteLink($h[i]$)
44: **end if**
45: **if** $g[i]$.type $==$ $DependsOn^{Partial}$ **then**
46: **for each** $\beta[j]$ **do**
47: **if** Dependency($\beta[j]$, $c[i]$) **then**
48: $\beta[j].DependsOn^{Partial} \leftarrow c[i]$
49: $c[i].NecessaryFor^{Partial} \leftarrow \beta[j]$
50: **end if**
51: **end for**
52: DeleteLink($h[i]$)
53: **end if**
54: **if** $g[i]$.type $==$ *NecessaryFor* **then**
55: **for each** $\beta[j]$ **do**
56: **if** Dependency($c[i]$, $\beta[j]$ **then** ▷ If $c[i]$ depends on information in $\beta[i]$ propagated from α.
57: $\beta[j]$.NecessaryFor $\leftarrow c[i]$
58: $c[i]$.DependsOn $\leftarrow \beta[j]$
59: **end if**
60: **end for**
61: DeleteLink($h[i]$)
62: **end if**

Algorithm 4 Decomposition Process - Part 2/2

```
63:         if g[i].type == NecessaryFor^Partial then
64:             for each β[j] do
65:                 if Dependency(c[i], β[j] then
66:                     β[j].NecessaryFor^Partial ← c[i]
67:                     c[i].DependsOn^Partial ← β[j]
68:                 end if
69:             end for
70:             DeleteLink(h[i])
71:         end if
72:         if g[i].type == AppliesTo then
73:             for each β[j] do
74:                 if Application(β[j], c[i]) then[6]                    ▷ If β[j] applies to c[i].
75:                     β[j].AppliesTo ← c[i]
76:                     c[i].SubjectToApplicationOf ← β[j]
77:                 end if
78:             end for
79:             DeleteLink(h[i])
80:         end if
81:         if g[i].type == SubjectToApplicationOf then
82:             for each β[j] do
83:                 if Application(c[i], β[j] then
84:                     β[j].SubjectToApplicationOf ← c[i]
85:                     c[i].AppliesTo ← β[j]
86:                 end if
87:             end for
88:             DeleteLink(h[i])
89:         end if
90:         if g[i].type == ConflictsWith then
91:             for each β[j] do
92:                 if Conflict(β[j], c[i]) then                     ▷ If β[j] conflicts with c[i].
93:                     β[j].ConflictsWith ← c[i]
94:                     c[i].ConflictsWith ← β[j]
95:                 end if
96:             end for
97:         end if ▷ The link h[i] is not deleted since the conflict between α and c[i] should still be indicated, even with α
   in the Inactive System.
98:     end for
99:     Deactivate(α)
100:    Return Success
101: end procedure
```

9.3.2 Textual Description

Artifacts may be broken into two, or more, artifacts through the decomposition process. This process is composed of two steps: generation of new links and rearrangement of existing links.

Generating Links

A group of n actors $\mathcal{A} = \{a_1, a_2, \ldots, a_n\}$, where $n \geq 1$, creates a Rationale for Decomposition P having an artifact α as its target. The artifact P should be homologated for the Decomposition Process to take place. Following the homologation of P, the r artifacts defined in P – the resulting artifacts of the decomposition – are created (each artifact will go through the Creation Process); let $\beta = \{\beta_1, \beta_2, \ldots, \beta_r\}$ be the set of these artifacts. For each $\beta_i \in \beta$, a *DecomposedFrom* link starting in β_i and having α as its destination is

[6] A function which, given two artifacts, returns if the first artifact depends on the second artifact.

generated. Let $\mathcal{B} = \{b_1, b_2, \ldots, b_q\}$ be the group of actors who created β. For each $b_j \in \mathcal{B}$, two links are generated: a *Decomposed* link starting in b_j and having α as its destination and a *DecomposedBy* link starting in α and having b_j as its destination. The successful homologation of each $\beta_i \in \beta$, done by a group of actors disjoint of $\mathcal{B}$, is necessary for the decomposition process to be complete. The parent artifact α and P are moved to the Inactive System following the homologation of all the artifacts in β.

Rearranging links

Rearrangement of links may be necessary after the replacement of α by the set of artifacts β. Every link starting, or having α as its destination, should be evaluated.

Let $G = \{g_1, g_2, \ldots, g_r\}$ be the set of links starting in α and let $\mathcal{C} = \{c_1, c_2, \ldots, c_r\}$ the set of destination artifacts of G. Let $H = \{h_1, h_2, \ldots, h_r\}$ be the set of opposite links to G, starting in C, where g_i is opposite to h_i.

For each *ReifiedFrom* link g_i, for each β_j which contains information propagated from c_i to α, two links are generated: a *ReifiedFrom* link g_i^* starting in β_j and having c_i as its destination and a *ReifiedTo* link h_i^* starting in c_i and having β_j as its destination. All the information from g_i and h_i is copied into g_i^* and h_i^*, respectively. No redirection is done as it may be necessary to generate multiple copies of h_i. Link h_i is deleted after the generation of the last h_i^* link.

For each *ReifiedTo* link g_i, for each β_j which contains information propagated to c_i, two links are generated: a *ReifiedTo* link g_i^* starting in β_j and having c_i as its destination and a *ReifiedFrom* link h_i^* starting in c_i and having β_j as its destination. All the information from g_i and h_i is copied into g_i^* and h_i^*, respectively. No redirection is done. Link h_i is deleted after the generation of the last h_i^* link.

For each *DependsOn* or *DependsOn*^{Partial} link g_i, for each β_j which depends on c_i, two links are generated: a *DependsOn* or *DependsOn*^{Partial} link g_i^* starting in β_j and having c_i as its destination and a *NecessaryFor* or *NecessaryFor*^{Partial} link h_i^* starting in c_i and having β_j as its destination. All the information from g_i and h_i is copied into g_i^* and h_i^*, respectively. No redirection is done. Link h_i is deleted after the generation of the last h_i^* link.

For each *NecessaryFor* or *NecessaryFor*^{Partial} link g_i, for each β_j which is necessary for c_i, two links are generated: a *NecessaryFor* or *DependsOn*^{Partial} link h_i^* link g_i^* starting in β_j and having c_i as its destination and a *DependsOn* or *DependsOn*^{Partial} link h_i^* starting in c_i and having β_j as its destination. All the information from g_i and h_i is copied into g_i^* and h_i^*, respectively. No redirection is done. Link h_i is deleted after the generation of the last h_i^* link.

For each *AppliesTo* link g_i, for each β_j which applies to c_i, two links are generated: an *AppliesTo* link g_i^* starting in β_j and having c_i as its destination and a *SubjectToApplicationOf* link h_i^* starting in c_i and having β_j as its destination. All the information from g_i and h_i is copied into g_i^* and h_i^*, respectively. No redirection is done. Link h_i is deleted after the generation of the last h_i^* link.

For each *SubjectToApplicationOf* link g_i, for each β_j which is subject to the application of c_i, two links are generated: a *SubjectToApplicationOf* link g_i^* starting in β_j and having

c_i as its destination and an *AppliesTo* link h_i^* starting in c_i and having β_j as its destination. All the information from g_i and h_i is copied into g_i^* and h_i^*, respectively. No redirection is done. Link h_i is deleted after the generation of the last h_i^* link.

For each *ConflictsWith* link g_i, for each β_j which contains information conflicting with c_i, two links are generated: a *ConflictsWith* link g_i^* starting in β_j and having c_i as its destination and a *ConflictsWith* link h_i^* starting in c_i and having β_j as its destination. All the information from g_i and h_i is copied into g_i^* and h_i^*, respectively. No redirection is done.

Classification of parts should be redone for each β_j; the classification of α into parts is kept unchanged.

9.4 Creation Process

Let α be an artifact created by a group of actors $\mathcal{A}$. Every newly created artifact starts in the Inactive System; therefore, α has to go through the Homologation Process to become part of the Active System; Rationale for Homologation or Rejection artifacts don't go through the Homologation Process, never being added to the Active System, since they are kept only for historical reasons. The successful homologation of α results in moving it from the Inactive System to the Active System.

9.4.1 Algorithm

Let α be an artifact created by a group of actors $\mathcal{A} = \{a_1, a_2, \ldots, a_n\}$, where $n \geq 1$. Let List of Artifacts ArtifactsReifiedFrom[] be the set of artifacts from which α was reified from. Let List of Artifacts ArtifactsAppliesTo[] be the set of artifacts which α applies to. Let List of Artifacts ArtifactsDependsOn[] be the set of artifacts which α depends on. Let List of Artifacts ArtifactsDependsOnPartial[] be the set of artifacts which depends on, but there other artifacts satisfying the same dependency. Let List of Artifacts ArtifactsNecessaryFor[] be the set of artifacts which α is necessary for. Let List of Artifacts ArtifactsNecessaryForPartial[] be the set of artifacts which is necessary for, but there other artifacts satisfying the same dependency. Let List of Artifacts ArtifactsConflictsWith[] be the set of artifacts which α conflicts with. Let Type RationaleType be the type of rationale α is, if α is a rationale artifact. Let Artifact RationaleTarget be the target artifact of α, if α is a rationale artifact. Let Artifact ArtifactDecomposedFrom be the artifact from which α was decomposed from. Let List of ArtifactsComprisedOf[] be the set of artifacts α is comprised of; i.e., artifact α should be classified into parts. Let Artifact RationaleForCreation be the Rationale for Creation which justifies, or justifies and described, the creation of α. Any of these parameters may be empty (NULL).

Algorithm 5 Creation Process - Part 1/2

```
1: procedure CreationProcess(Artifact α, Group of Actors 𝒜, List of Artifacts ArtifactsReifiedFrom[], Type RationaleType, Artifact RationaleTarget, Artifact ArtifactDecomposedFrom, List of Artifacts ArtifactsAppliesTo[], List of Artifacts ArtifactsDependsOn[], List of Artifacts ArtifactsDependsOnPartial[], List of Artifacts ArtifactsNecessaryFor[], List of Artifacts ArtifactsNecessaryForPartial[], List of Artifacts ArtifactsConflictsWith[], List of ArtifactsComprisedOf[], Artifact RationaleForCreation)
2:     for each a[i] ∈ 𝒜 do
3:         α.CreatedBy[i] ← a[i]
4:         a[i].Created ← α
5:     end for
6:     if ArtifactsReifiedFrom[] IS NOT NULL then                ▷ Is a reification of other Artifacts.
7:         for each ArtifactsReifiedFrom[i] do
8:             α.ReifiedFrom[i] ← ArtifactsReifiedFrom[i]   ▷ The opposite link is created in the Homologation Process.
9:         end for
10:    end if
11:    if RationaleType IS NOT NULL then                         ▷ Is a Rationale.
12:        α.Type ← RationaleType
13:        switch α.Type do
14:            case RM
15:                α.DefinesModificationOf ← RationaleTarget
16:            case RR
17:                α.DefinesRemovalOf ← RationaleTarget
18:            case RD
19:                α.DefinesDecompositionOf ← RationaleTarget
20:            case RAc
21:                α.DefinesActivationOf ← RationaleTarget
22:            case RAp
23:                α.JustifiesApplicationOf ← RationaleTarget
24:        end switch
25:    end if
26:    if ArtifactDecomposedFrom IS NOT NULL then                ▷ Is a decomposition of an Artifact.
27:        α.DecomposedFrom ← ArtifactDecomposedFrom
28:    end if
29:    if ArtifactsAppliesTo[] IS NOT NULL then                  ▷ Should be applied to other Artifacts.
30:        for each ArtifactsAppliesTo[i] do
31:            α.AppliesTo[i] ← ArtifactsAppliesTo[i]
32:        end for
33:    end if
34:    if ArtifactsDependsOn[] IS NOT NULL then                  ▷ Depends on other Artifacts.
35:        for each ArtifactsDependsOn[i] do
36:            α.DependsOn[i] ← ArtifactsDependsOn[i]
37:        end for
38:    end if
39:    if ArtifactsDependsOnPartial[] IS NOT NULL then
40:        for each ArtifactsDependsOnPartial[i] do
41:            α.DependsOnPartial[i] ← ArtifactsDependsOnPartial[i]
42:        end for
43:    end if
44:    if ArtifactsNecessaryFor[] IS NOT NULL then               ▷ Is necessary for other Artifacts.
45:        for each ArtifactsNecessaryFor[i] do
46:            α.NecessaryFor[i] ← ArtifactsNecessaryFor[i]
47:        end for
48:    end if
49:    if ArtifactsNecessaryForPartial[] IS NOT NULL then
50:        for each ArtifactsNecessaryForPartial[i] do
51:            α.NecessaryForPartial[i] ← ArtifactsNecessaryForPartial[i]
52:        end for
53:    end if
54:    if ArtifactsConflictsWith[] IS NOT NULL then              ▷ Conflicts with other Artifacts.
55:        for each ArtifactsConflictsWith[i] do
56:            α.ConflictsWith[i] ← ArtifactsConflictsWith[i]
57:            ArtifactsConflictsWith[i].ConflictsWith ← α
58:        end for
59:    end if
```

Algorithm 5 Creation Process - Part 2/2

```
60:     if ArtifactsComprisedOf[] IS NOT NULL then                ▷ Identifying components of α.
61:         for each ArtifactsComprisedOf[i] do
62:             α.ComprisedOf[i] ← ArtifactsComprisedOf[i]
63:             ArtifactsComprisedOf[i].PartOf ← α
64:         end for
65:     end if
66:     if RationaleForCreation IS NOT NULL then     ▷ If it was necessary to justify the creation of α.
67:         α.CreationDefinedBy ← RationaleForCreation
68:         RationaleForCreation.DefinesCreationOf ← α
69:     end if
70: end procedure
```

9.4.2 Textual Description

It is necessary to generate traceability links for a newly created artifact. Let α be an artifact created by a group of actors $\mathcal{A} = \{a_1, a_2, \ldots, a_n\}$, where $n \geq 1$. For each $a_i \in \mathcal{A}$, two links are generated: a *CreatedBy* link starting in α and having a_i as its destination and a *Created* link starting in a_i and having α as its destination. It may be necessary to generate more links, as follows.

If α is a reification of other artifacts it will be necessary to generate links. Let $\beta = \{\beta_1, \beta_2, \ldots, \beta_m\}$ be the set of m artifacts reified into α. A set of links $E = \{e_1, e_2, \ldots, e_m\}$ is generated, where each $e_i \in E$, $1 \leq i \leq m$, is a *ReifiedFrom* link starting in α and having $\beta_i \in \beta$ as its destination.

If α is a rationale type artifact, it will be necessary to generate a link. Let β be the target of α. If α is an RM, a *DefinesModificationOf* link is generated starting in α and having β as its destination. If α is an RD, a *DefinesDecompositionOf* link is generated starting in α and having β as its destination. If α is an RR, a *DefinesRemovalOf* link is generated starting in α and having β as its destination. If α is an RAc, a *DefinesActivationOf* link is generated starting in α and having β as its destination. If α is an RAp, a *JustifiesApplicationOf* link is generated starting in α and having β as its destination.

If α is a decomposition from another artifact, it will be necessary to generate a link. Let β be the parent of α; a *DecomposedFrom* link is generated starting in α and having β as its destination.

If α should be applied to other artifacts, it will be necessary to generate links. Let $\beta = \{\beta_1, \beta_2, \ldots, \beta_m\}$ be the set of m artifacts which α should be applied to; a set of links $E = \{e_1, e_2, \ldots, e_m\}$ is generated where each $e_i \in E$, $1 \leq i \leq m$, is an *AppliesTo* link starting in α and having $\beta_i \in \beta$ as its destination.

If α depends on other artifacts, it will be necessary to generate links. Let $\beta = \{\beta_1, \beta_2, \ldots, \beta_m\}$ be the set of m artifacts necessary for α; a set of links $E = \{e_1, e_2, \ldots, e_m\}$ is generated where each $e_i \in E$, $1 \leq i \leq m$, is a *DependsOn* link starting in α and having $\beta_i \in \beta$ as its destination. If there is more than one artifact satisfying α, given a certain dependency, a $DependsOn^{Partial}$ link must be created instead of a *DependsOn* link.

If α is necessary for other artifacts, it will be necessary to generate links. Let $\beta = \{\beta_1, \beta_2, \ldots, \beta_m\}$ be the set of m artifacts dependent on α; a set of links $E = \{e_1, e_2, \ldots, e_m\}$ is generated where each $e_i \in E$, $1 \leq i \leq m$, is a *NecessaryFor* link starting in α and having

$\beta_i \in \beta$ as its destination. If there are other artifacts satisfying $\beta_i \in \beta$, given a certain dependency, a *NecessaryFor*^{Partial} link must be created instead of a *NecessaryFor* link.

An artifact α may have dependencies which are not fulfilled by any artifact in the Active System. These dependencies may be fulfilled – after its homologation – through the creation, modification, or activation of artifacts.

If α has conflicts with other artifacts in the Active or in the Inactive System, it will be necessary to generate links. Let $\beta = \{\beta_1, \beta_2, \ldots, \beta_m\}$ be the set of m artifacts which conflict with α; two sets of links are generated: $E = \{e_1, e_2, \ldots, e_m\}$ and $F = \{f_1, f_2, \ldots, f_m\}$. Each $e_i \in E$, $1 \leq i \leq m$, is a *ConflictsWith* link starting in α and having $\beta_i \in \beta$ as its destination. Each $f_i \in F$ is a *ConflictsWith* link starting in $\beta_i \in \beta$ and having α as its destination.

If it is necessary to identify α's components, let $\beta = \{\beta_1, \beta_2, \ldots, \beta_m\}$ be the set of α's components; two sets of links are generated: $E = \{e_1, e_2, \ldots, e_m\}$ and $F = \{f_1, f_2, \ldots, f_m\}$. Each $e_i \in E$, $1 \leq i \leq m$, is a *ComprisedOf* link starting in α and having $\beta_i \in \beta$ as its destination. Each $f_i \in F$ is a *PartOf* link starting in $\beta_i \in \beta$ and having α as its destination.

If it is necessary to justify α's creation, let β be an RC; two links are generated: a *CreationDefinedBy* link starting in α and having β as its destination and a *DefinesCreationOf* link starting in β and having α as its destination.

Every newly created artifact starts in the Inactive System; it has to go through the Homologation Process to become part of the Active System. Rationale for Homologation or Rejection artifacts don't go through the Homologation Process, never being added to the Active System, since they are kept only for historical reasons.

9.5 Removal Process

The Removal Process moves an artifact from the Active System to the Inactive System. Artifacts are never destroyed, they are kept for historical reasons or for possible future activations. Let α be an artifact which goes through the Removal Process. A group of actors $\mathcal{A}$ creates a Rationale for Removal P having α as its target. The rationale P should be homologated for the removal process to take place. After the successful homologation of P, a group of actors $\mathcal{B}$ performs the removal described in P by deactivating α. The removal process may require changes on artifacts related to the removed artifact; for instance, the removal of a code fragment, e.g. a function, may require removing calls in other artifacts. The artifact P is then moved to the Inactive System.

9.5.1 Algorithm

A group of actors $\mathcal{A} = \{a_1, a_2, \ldots, a_n\}$, where $n \geq 1$, creates a Rationale for Removal P having α as its target. The artifact P should be homologated for the Removal Process to take place. A group of actors $\mathcal{B}$ performs the removal of α.

Algorithm 6 Removal Process

```
1: procedure RemovalProcess(Artifact α, Group of Actors 𝒜, Group of Actors ℬ, Rationale for Removal P)
2:     ACTION: ℬ performs the removal of α as described in P.[7]
3:     Deactivate(α)
4:     for each b[i] ∈ ℬ do
5:         α.RemovedBy[i] ← b[i]
6:         b[i].Removed ← α
7:     end for
8:     Deactivate(P)
9: end procedure
```

9.5.2 Textual Description

Artifacts may be moved from the Active System to the Inactive System through the Removal Process. A group of n actors $\mathcal{A} = \{a_1, a_2, \ldots, a_n\}$, where $n \geq 1$, creates a Rationale for Removal P having α as its target. The artifact P should be homologated for the Removal Process to take place.

A group of n actors $\mathcal{B} = \{b_1, b_2, \ldots, b_q\}$, where $q \geq 1$, performs the removal described in P by deactivating α. Two sets of links are generated: $E = \{e_1, e_2, \ldots, e_q\}$ and $F = \{f_1, f_2, \ldots, f_q\}$. Each $e_i \in E$, $1 \leq i \leq q$, is a *RemovedBy* link starting in α and having $b_i \in \mathcal{B}$ as its destination. Each $f_i \in F$ is a *Removed* link starting in $b_i \in \mathcal{B}$ and having α as its destination. The artifact P is then moved to the Inactive System.

The removal process may require changes on artifacts related to the removed artifact; for instance, the removal of a code fragment (e.g. a function) may require removing calls in other artifacts.

9.6 Activation Process

The Activation Process enables the activation of inactive artifacts which were previously rejected during the Homologation Process or removed by the Removal Process. Let α be an artifact which goes through the Activation Process. A group of actors $\mathcal{A}$ creates a Rationale for Activation P having α as its target. The artifact P should be homologated for the Activation Process to take place; i.e., a group of actors disjoint from $\mathcal{A}$ will decide if α should be part of the Active System. Following the homologation of P, α is moved into the Active System. The artifact P is then moved out to the Inactive System.

The activation of an artifact may be contingent upon its modification; i.e., the modification makes the artifact useful for the Active System. In this case, the rationale P not only justifies the activation of α but also describes the modifications to be done on α. The artifact P should be homologated for the activation process to take place. Following the homologation of P, a Rationale for Modification Q is created containing the modifications described in P. It is optional to homologate Q; it may be added to the Active System soon after its creation or it may go through the Homologation Process to ensure it contains the exact information described in P. The artifact α goes through the Modification Process

[7] The removal may involve more than just deactivating the Artifact; e.g., the removal of a function may require removing calls in other artifacts. This example would generate a sequence of modifications in related artifacts.

started by Q. A successful modification and homologation of the new version of α results in the activation of α and the deactivation of P and Q.

9.6.1 Algorithm

A group of actors $\mathcal{A} = \{a_1, a_2, \ldots, a_n\}$, where $n \geq 1$, creates a Rationale for Activation P having α as its target. The artifact P should be homologated for the activation process to take place. If P indicates a modification to be performed on α, let Group of Actors $\mathcal{B}$ and Group of Actors $\mathcal{C}$ be the groups of actors which performs and homologates the modification, respectively.

Algorithm 7 Activation Process - Part 1/2

```
1: procedure ActivationProcess(Artifact α, Group of Actors 𝒜, Group of Actors ℬ, Group of Actors 𝒞, Rationale for
   Activation P)
2:     New Artifact αToBeActivated
3:     if P.hasModification == True then                ▷ The activation is contingent upon a modification.
4:         CreationProcess(Q, 𝒜, NULL, "RM", α, NULL, NULL, NULL, NULL, NULL, NULL, NULL)    ▷ Creates the
   Rationale for Modification Q using information from P.
5:         Activate(Q)      ▷ Alternatively, Q may go through the Homologation Process to ensure it contains the exact
   information described in P.
6:         α¹ ← ModificationProcess(α, ℬ, 𝒞, Q)      ▷ ℬ performs the modification and 𝒞 homologates the modification.
7:         if α¹ != NULL then
8:             αToBeActivated ← α¹
9:             Activate(αToBeActivated)
10:            Deactivate(P)
11:            Deactivate(Q)
12:        else
13:            Return NULL
14:        end if
15:    else
16:        αToBeActivated ← α
17:        Activate(αToBeActivated)
18:    end if
19:    if P.ListOfDependents IS NOT NULL then          ▷ αToBeActivated is being activated to fulfill dependencies.
20:        for each ListOfDependents[i] do
21:            αToBeActivated.NecessaryFor = ListOfNewDependencies[i]
22:            ListOfDependents[i].DependsOn ← αToBeActivated
23:        end for
24:    end if
25:    if αToBeActivated.NumberOfUnfulfilledDependencies > 0 then  ▷ αToBeActivated has unfulfilled dependencies.
26:        ListOfNewDependencies[] ← GetsListOfNewDependencies(αToBeActivated)
27:        for each ListOfNewDependencies[i] do
28:            if ListOfNewDependencies[i].partialDependency == False then      ▷ Is the only active artifact capable of
   satisfying this dependency.
29:                αToBeActivated.DependsOn = ListOfNewDependencies[i]
30:                ListOfNewDependencies[i].NecessaryFor ← αToBeActivated
31:            else
32:                αToBeActivated.DependsOnPartial = ListOfNewDependencies[i]
33:                ListOfNewDependencies[i].NecessaryForPartial ← αToBeActivated
34:            end if
35:            αToBeActivated.NumberOfUnfulfilledDependencies–
36:        end for
37:        if αToBeActivated.NumberOfUnfulfilledDependencies > 0 then
38:            ACTION: Create, modify, or activate Artifacts to fulfill the dependencies.
39:        end if
40:    end if
```

Algorithm 7 Activation Process - Part 2/2

```
41:     if αToBeActivated.hasReifiedToLinks == True then        ▷ Not the final artifact in a reification sequence.
42:         ACTION: Create, modify, or activate Artifacts to propagate the information to the Active System.
43:     end if
44:     if αToBeActivated.hasConflictsWithLinks == True then
45:         for each αToBeActivated.ConflictsWith[i] do
46:             ListOfActiveArtifacts = IsActive(αToBeActivated.ConflictsWith[i])        ▷ Checks if any of the conflicting
        artifacts are in the Active System.
47:         end for
48:         if ListOfActiveArtifacts IS NOT NULL then
49:             ACTION: Modify or remove the conflicting artifacts from the Active System to solve the conflicts.
50:         end if
51:     end if
52:     Return Success
53: end procedure
```

9.6.2 Textual Description

Inactive artifacts which were previously rejected during the homologation process, or removed from the Active System by the Removal Process, may become active through the Activation Process.

A group of n actors $\mathcal{A} = \{a_1, a_2, \ldots, a_n\}$, where $n \geq 1$, creates a Rationale for Activation P having α as its target. The artifact P should be homologated for the activation process to take place; i.e., a group of actors disjoint from $\mathcal{A}$ will decide if α should be part of the Active System. Following the homologation of P, α is moved into the Active System. The artifact P is then moved out to the Inactive System.

9.6.2.1 About the Rationale for Activation

The Rationale for Activation stores the reason for the activation of the target artifact, therefore, it may also store information identifying artifacts related to the target artifact; e.g., if an artifact is being activated to fulfill dependencies in the Active System, the Rationale for Activation will have the list of Artifacts which the target artifact is necessary for.

An activation may also require the modification of the target artifact; in this case, the Rationale for Activation stores a complete Rationale for Modification in its description.

9.6.2.2 Activation Contingent upon Modification

The activation of an artifact may be contingent upon its modification; i.e., the modification makes the artifact useful for the Active System.

A group of n actors $\mathcal{A} = \{a_1, a_2, \ldots, a_n\}$, where $n \geq 1$, creates an RAc P having α as its target. The RAc P describes a modification to be done on α. The artifact P should be homologated for the activation process to take place. Following the homologation of P, an RM Q is created containing the modifications described in P. It is optional to homologate Q; it may be added to the Active System soon after its creation or it may go through the homologation process to ensure it contains the exact information described in P. The artifact α goes through the modification process started by Q. A successful modification results in the activation of α and the deactivation of P and Q.

Generating Links

It may be necessary to generate links after the activation of α, as follows.

If α is being added to the Active System to fulfill dependencies, links should be generated. Let $\beta = \{\beta_1, \beta_2, \ldots, \beta_m\}$ be the set of active artifacts having unfulfilled dependencies which will be fulfilled by α. Two sets of links are generated: $E = \{e_1, e_2, \ldots, e_m\}$ and $F = \{f_1, f_2, \ldots, f_m\}$. Each $e_i \in E$, $1 \leq i \leq m$, is a *NecessaryFor* link starting in α and having $\beta_i \in \beta$ as its destination. Each $f_i \in F$ is a *DependsOn* link starting in $\beta_i \in \beta$ and having α as its destination.

If α has unfulfilled dependencies, these should be addressed. If there are artifacts in the Active System fulfilling all, or part of, α dependencies, the corresponding links should be generated. Let $\beta = \{\beta_1, \beta_2, \ldots, \beta_m\}$ be the set of active artifacts capable of fulfilling dependencies of α; two sets of links are generated: $E = \{e_1, e_2, \ldots, e_m\}$ and $F = \{f_1, f_2, \ldots, f_m\}$. Each $e_i \in E$, $1 \leq i \leq m$, is a *DependsOn* or $\textit{DependsOn}^{Partial}$ link starting in α and having $\beta_i \in \beta$ as its destination. Each $f_i \in F$ is a *NecessaryFor* or $\textit{NecessaryFor}^{Partial}$ link starting in $\beta_i \in \beta$ and having α as its destination. If there are still unfulfilled dependencies, artifacts must be created, modified, or activated to fulfill the remaining dependencies.

If α is not the final artifact in a reification sequence, having *ReifiedTo* links starting from it, it has information which should be propagated. Since α is in the Inactive System, the artifacts having their propagated information are also in the Inactive System; hence, artifacts must be created, modified, or activated to propagate the information contained in α to the Active System.

If α has *ReifiedFrom* links starting from it, no action is necessary since the destination artifacts are in the Inactive System. These links will be kept for historical reasons.

If α has *NecessaryFor* or $\textit{NecessaryFor}^{Partial}$ links starting from it, having artifacts in the Inactive System as destination, no action is necessary. These links will be kept for historical reasons; they may be useful if any of the destination artifacts are activated in the future.

If α has *ConflictsWith* links starting from it, having artifacts in the Active System as destination, it will be necessary to modify or remove the destination artifacts to solve the conflicts.

9.7 Application Process

Certain artifacts may be applied to other artifacts. For instance, the application of a test case on a part of code. Let α be an artifact which goes through the Application Process. A group of actors $\mathcal{A}$ applies α on its targets. The application of α may result in the creation of one, or more, artifacts; e.g., running a test case may create a log file recording the outcome of the test.

9.7.1 Algorithm

A group of actors $\mathcal{A} = \{a_1, a_2, \ldots, a_n\}$, where $n \geq 1$, applies an artifact α on its targets.

Algorithm 8 Application Process

```
1: procedure APPLICATIONPROCESS(Artifact α, Group of Actors 𝒜, Rationale for Application P)
2:     ACTION: α is applied on its targets. If Artifacts are generated, these are assigned to ListOfArtifacts[].
3:     for each a[i] ∈ 𝒜 do
4:         α.AppliedBy[i] ← a[i]
5:         a[i].Applied ← α
6:     end for
7:     if ListOfArtifacts IS NOT NULL then        ▷ One or more Artifacts were created during the application of α.
8:         for each ListOfArtifacts[i] do
9:             α.ApplicationProduces[i] ← ListOfArtifacts[i]
10:            ListOfArtifacts[i].ProducedByApplicationOf ← α
11:        end for
12:    end if
13: end procedure
```

9.7.2 Textual Description

Certain artifacts may be applied to other artifacts. The application of an artifact may result in the creation of one, or more, artifacts. For instance, running a test case may create a log file recording the outcome of the test.

A group of n actors $\mathcal{A} = \{a_1, a_2, \ldots, a_n\}$, where $n \geq 1$, applies an artifact α on its targets. After the application of α, two sets of links are generated: $E = \{e_1, e_2, \ldots, e_n\}$ and $F = \{f_1, f_2, \ldots, f_n\}$. Each $e_i \in E$, $1 \leq i \leq n$, is an *AppliedBy* link starting in α and having $a_i \in \mathcal{A}$ as its destination. Each $f_i \in F$ is an *Applied* link starting in $a_i \in \mathcal{A}$ and having α as its destination.

If the application of α results in the creation of one, or more, artifacts: let $\beta = \{\beta_1, \beta_2, \ldots, \beta_m\}$ be the set of m artifacts created as a result of α's application; two sets of links are generated: $G = \{g_1, g_2, \ldots, g_m\}$ and $H = \{h_1, h_2, \ldots, h_m\}$. Each $g_i \in G$, $1 \leq i \leq m$, is an *ApplicationProduces* link starting in α and having $\beta_i \in \beta$ as its destination. Each $h_i \in H$, is a *ProducedByApplicationOf* link starting in $\beta_i \in \beta$ and having α as its destination.

9.8 Processes and Permissions

For every action performed by actors in each process, it is assumed the actors involved have the necessary permission to carry out the action. For instance, if a group of actors modify an artifact it is presumed they have the permission to do so.

9.8.1 Creating and Updating Permissions

It may be necessary to automatically create or update permissions whenever a process is performed. General guidelines when assigning permissions to artifacts, for each process, are shown next.

9.8.1.1 Creation Process

New permissions are assigned to the newly created artifact; the Rules Space from a project will have the set of rules defining which permissions each new artifact should receive. A

basic rule would connect permissions between actors and the new artifact, since actors must be able to manipulate the new artifact which was added to the System Space.

9.8.1.2 Modification Process

The new version inherits the whole, or partial, set of permissions from the old version. New permissions may also be assigned to the new version; i.e., a modified artifact may have new characteristics which demand the inclusion of new actors to handle its added modifications, hence, a new group of actors will receive permissions to manipulate it.

9.8.1.3 Decomposition Process

Each of the new artifacts will be assigned the whole, or partial, set of permissions from its parent artifact. Some permissions may be related to specific information in the parent artifact, hence, these should be inherited only by the artifacts to which this information was propagated.

9.8.1.4 Homologation, Removal, Activation, and Application Processes

Permissions are unchanged for the processes of homologation, removal, activation and application.

9.9 Change Impact Analysis and Processes

Traceability processes are very useful for change impact analysis because, while providing instructions to avoid issues of traceability consistency and system consistency, they reveal the consequences of actions.

A process may be evaluated as a decision tree; a change may result in multiple paths, each leading to a leaf node. A leaf node represents the consequences of the change. For instance, the modification of an artifact α having a DependsOn link connected to artifact β_1 and a ReifiedFrom link connected to artifact β_2 generates two paths (see Section 9.1.3): in the DependsOn path, if β_1 is not necessary after the proposed modification, the leaf node instructs us to delete traceability links and evaluate the usefulness of β_1 and maybe remove it; in the ReifiedFrom path, if α does not contain information from β_2 after the proposed modification, the leaf node instructs us to modify or remove β_2, depending if it propagates, or not, information to other artifacts, respectively.

The assortment of leafs, obtained by following each path caused by a change, provides the complete set of consequences of an action. This set can be used to evaluate the cost of a change in the System Space.

Traceability processes, together with traceability links, enable the possibility of evaluating the cost of a change, and consequently cancelling a change – having costs which outweigh its benefits – before its implementation; thus, these elements are very useful for change impact analysis.

Chapter 10

Using the Metamodel

In this chapter, we illustrate a partial application of the Metamodel by using it on requirements of a course management system found in [25].[1] We found a few problems with these requirements; we will discuss possible solutions in this chapter.

This chapter is structured as follows: Section 10.1 shows the issues found and how to fix them by using the Metamodel; and Section 10.2 provides a few closing remarks about this chapter.

10.1 Conflicts Between Requirements

Issues were found when identifying relations between certain requirements; there are ConflictsWith links connecting the requirements R17 and R74, and requirements R17 and R18. The requirement R74 determines that only lecturers are allowed to create teams; however, the requirement R17 allows students to create teams. Requirement R18 provides, in a more detailed way, the same information as requirement R17; i.e., there is redundant information in the project. Since every requirement in the document is assumed active, there are inconsistencies in the System Space; these requirements can not be in the Active System at the same time (see Definition 8.2.6 in Chapter 8).

Table 10.1 shows the description of each relevant requirement.

[1] The complete list of requirements is available at http://bit.ly/reqcms.

Table 10.1: Selected Requirements

Requirement Number	Requirement Description
R07	The system shall provide a messaging system.
R16	The System shall allow sending messages to individuals, teams or all course participants at once.
R17	The system shall allow students to create teams.
R18	Teams are created by students inviting other students in the same course using the messaging system.
R74	The system shall allow only lecturers to create new teams.
R75	The system shall allow lecturers to insert students into teams.
R76	The system shall allow lecturers to remove students from teams.
R77	The system shall allow lecturers to delete teams.
R78	The system shall allow lecturers to assign (assistant) lecturers to teams.
R79	The system shall allow lecturers to name and rename teams.

First, let's identify relevant relations; Table 10.2 shows the pairs of traceability links connecting the requirements. The first and the third column show the requirement number; the second column shows the pair of traceability links connecting the requirements from the first and second column.

Table 10.2: Requirements - Traceability Links.

Requirement Number	Traceability Link Pair	Requirement Number
R07	NecessaryFor/DependsOn	R16, R18
R16	NecessaryFor/DependsOn[2]	R18
R17	ConflictsWith/ConflictsWith	R18, R74
R17	NecessaryFor^{Partial}/DependsOn^{Partial}	R75, R76, R77, R78, R79
R18	ConflictsWith/ConflictsWith	R74
R18	NecessaryFor^{Partial}/DependsOn^{Partial}	R75, R76, R77, R78, R79
R74	NecessaryFor^{Partial}/DependsOn^{Partial}	R75, R76, R77, R78, R79

There are NecessaryFor/DependsOn pairs of links between requirements R07 and R16, and between requirements R07 and R18; a messaging system must exist to enable sending messages to individuals and to invite students to a team. There is a NecessaryFor/DependsOn pair of links between requirements R16 and R18; a student must be allowed to send messages to invite other students. Requirement R18 also allows students to create teams and details how they can do it; thus, there is a ConflictsWith pair of links between R17 and R18 indicating redundant information. There

[2] Invitation is considered a message.

are ConflictsWith links between R74 and R17 since they are contradictory; requirement R74 determines that only lecturers should be allowed to create teams and R17 allows students to create teams. ConflictsWith links also exist between R74 and R18, for the same reasons there are ConflictsWith links between R74 and R17. There are NecessaryFor^{Partial}/DependsOn^{Partial} pairs of links between requirements R74 and requirements R75 to R79; student teams must exist to be managed by the lecturer. Analogously, there are NecessaryFor^{Partial}/DependsOn^{Partial} pairs of links between requirements R17 and requirements R75 to R79, and between requirements R18 and requirements R75 to R79. These are partial links because three requirements – R17, R18, and R74 – ensure the existence of student teams.

There are no Accountability or Permission links since there are no identification of actors in the document. There are no Evolution links of the reification type since there is only one activity being represented by the artifacts. Evolution links of the modification type, Characterize Action links, and Action Outcome links could exist if the document was created in the context of the Metamodel. There are no Composition links identified in this particular set of requirements.

The conflicts must be solved; there are several ways to fix these inconsistencies. Concerning the conflict between requirements R17 and R18, a simple solution would be to remove requirement R17; i.e., keeping the requirement which provides more information, comparatively. This action also solves the conflict between R17 and R74. Given this set of requirements, there are no necessary actions resulting from removing R17. After removing requirement R17, the conflict between R18 and R74 must be solved. Let's evaluate possible solutions for the remaining conflict

10.1.1 Removing Requirement R18

A possible solution is to remove requirement R18, disallowing students from creating teams. A Rationale for Removal is created having R18 as its target; this Rationale has to be homologated for the removal to occur. The process "Assessing the Impact of Homologating a Rationale for Modification or a Rationale for Removal" is used to evaluate possible consequences and solutions for this action (see Section 9.1.3 in Chapter 9). A Rationale for Removal is being assessed, so we go from line 2 of Algorithm 2 to line 99; there are DependsOn links starting in R18 (line 103). Requirement R18 depends on requirements R07 and R16; however, since these requirements do not exist exclusively to fulfill R18's dependency, no action is necessary (line 104); the messaging system has other uses besides inviting students to teams, and students should be able to send messages even if they are not allowed to create teams. There are NecessaryFor^{Partial} links starting in R18. The assessment process determines that no action is necessary; there are other artifacts to satisfy the same dependency, given it is a partial link (fifth paragraph of the textual description of Algorithm 2).

10.1.2 Modifying Requirement R74

Another solution is to modify requirement R74 to allow non-lecturers to create teams. The word “only” would be deleted from the requirement description. However, the project would then allow lecturers to create empty teams and students would be able to create teams by inserting other students into them. A more elegant solution is to modify R74 by removing the word “only”, adding students to its description, and removing requirement R18. This solution is also simpler if we decide to allow students to perform the team management activies specified by requirements R75, R76, R77, and R79. However, this would generate ConflictsWith links between requirements R75 and R18; both would contain the information that students may insert other students into teams.

A Rationale for Modification is created having R74 as its target; this Rationale has to be homologated for the modification to occur. The process “Assessing the Impact of Homologating a Rationale for Modification or a Rationale for Removal” is used. There are NecessaryFor^{Partial} links starting in R74, having requirements R75–R79 as their destination; however, the modification does not affect the dependency relation between R74 and these requirements; thus, no action is necessary (line 14 of Algorithm 2). The modified requirement R74 is shown next.

R74	The system shall allow lecturers and students to create new teams.

Requirement R18 must be removed; this action was described in Section 10.1.1. This leads to every NecessaryFor^{Partial}/DependsOn^{Partial} pair of links connected to requirement R74 to be treated as non-partial; there are no other partial pair of links in the Active System representing the same information.

Requirements R75, R76, R77, and R79 deal with team management; it makes sense to allow students to perform these tasks, given that they are allowed to create teams. In this case, Rationales for Modification should be created targeting these requirements. The modification would add students as agents in these requirements. The modified requirements are shown next.

R75	The system shall allow lecturers and students to insert students into teams.
R76	The system shall allow lecturers and students to remove students from teams.
R77	The system shall allow lecturers and students to delete teams.
R79	The system shall allow lecturers and students to name and rename teams.

This modification generates a conflict, as mentioned previously: the new version of requirement R75 is in conflict with requirement R18. Lines 81 to 86 of Algorithm 1 (Homologation Process) determine the generation of ConflictsWith between requirements R75 and R18. Since requirement R18 was removed, there are no conflicts between artifacts in the Active System; thus, no more actions are necessary.

10.1.3 Removing Requirement R18 and Requirement R74

A reestructuring of the requirements, to improve their quality and solve the conflicts, could lead to removing both requirements R18 and R74; however, students teams should still exist and someone should be able to create them.

First, requirement R18 is removed; this was described in Section 10.1.1. A Rationale for Removal is created having R74 as its target; this Rationale has to be homologated for the removal to occur. Given the removal of requirements R17 and R18, the NecessaryFor^{Partial}/DependsOn^{Partial} pairs of links connecting requirement R74 to requirements R75 to R79 should be treated as non-partial.

The Removal Process will happen after the homologation of the Rationale for Removal having requirement R74 as its target. Its homologation should be assessed by Algorithm 2. Requirement R74 is necessary for requirements R75 to R79. Line 101 proposes three alternatives: modify R75–R79 to end their dependency to requirement R74, remove requirements R75–R79 from the Active System, or create/modify/activate artifacts to satisfy R75–R79. These requirements will not be modified or removed. Thus, a new artifact will be created containing the information from requirements R17, R18, and R74; i.e., an artifact will be created to satisfy requirements R75–R79.

After the removal of requirement R74, a Rationale for Creation is created containing the following information: (i) it should satisfy R75 to R79 by allowing the creation of teams, (ii) it should allow lecturers and students to create teams, and (iii) students are added to teams by invitation (students) or by being added manually (lecturers). Since requirement R75 already provides (iii) for lecturers, this requirement will have to be removed to avoid a redundancy conflict. The new requirement is shown next.

R74-2	The system shall allow lecturers and students to create new teams; teams are filled by students invited by the team creator (if student) or by direct assignment (if lecturer).

Traceability links must be created: lines 44–48 of Algorithm 5 (Creation Process) determine the generation of four NecessaryFor links starting in requirement R74-2 and having requirements R76, R77, R78, and R79, as their destination. The opposite links are generated during the homologation of requirement R74-2; lines 57–61 of Algorithm 1 (Homologation Process) determine the generation of four DependsOn links starting in requirements R76, R77, R78 and R79, and having requirement R74-2 as their destination.

Also, lines 54–59 of Algorithm 5 determine the generation of a ConflictsWith pair of links between requirement R74-2 and requirement R75; both requirements describe the insertion of students into teams. The subsequent homologation of requirement R74-2 identifies the conflict and determines the removal of requirement R75 (lines 78–80 of Algorithm 1). Therefore, a Rationale for Removal is created having requirement R75 as its target; after its homologation the artifact is removed from the Active System.

10.2 Closing Remarks

This chapter intends to illustrate the use of the Metamodel in a project. The Metamodel may be used to fix existing issues, as it was used here; however, the issues found in these artifacts could be avoided by using the Metamodel from the beginning of this project. These conflicts would be identified previously, during the Creation Process of the related artifacts, avoiding the generation of conflicts altogether.

These redundancy conflicts resulted in the generation of partial links; it may seem that partial links occur only whenever there is redundant information. However, it is not always the case; distinct sets of blocks of information may satisfy the same artifact. For instance, a contradictory information conflict may generate partial links of dependency. Consider this example provided in Chapter 3: two distinct requirements require the project to use exclusively IIS servers and Apache Servers, respectively. These requirements provide contradictory information in which a third requirement is dependent on; thus, two NecessaryFor^{Partial}/DependsOn^{Partial} pairs of links would connect these three requirements.

Finally, this chapter also shows how the Metamodel enables the comparison of different solutions; the impact of a change may be assessed beforehand, enabling an informed decision regarding changes. In this case, we have proposed three different solutions to solve the conflicts found; each solution could be evaluated by itself, or comparatively, before being implemented.

Part IV

Conclusion

Chapter 11

Conclusions, Limitations, and Future Work

We have identified a number of issues in current traceability models, such as: lack of description of link types and artifact types; lack of mechanisms to ensure consistency; missing link types for modeling common relations; and missing common activities of software development processes. These issues make most models incomplete or even unusable; for instance, link types lacking clear descriptions can not be mapped into relations in a project, and lack of mechanisms result in erroneous – and consequently unusable – traceability information after changes.

To address these issues we have devised a traceability Reference Model to support the creation, and evaluation, of traceability models. We identified the basic elements of traceability, basic properties, and basic sets of these elements. For instance, we have noticed how actions are intrinsically connected to link types and processes; thus, establishing the actions being contemplated by a model is essential to define these elements. We have defined the most common actions in software development and used them in the construction of sets of link types and processes for traceability. We have also established basic properties for the sets of link types and artifact types; these are useful when creating new sets or when evaluating existing sets.

The Reference Model was used to build a Metamodel for traceability; the Metamodel implements the Reference Model and expands on what it defines. The Metamodel is composed of a detailed conceptual model describing and organizing the elements of traceability; a set of artifact types to represent the most common activities of development or to record valuable traceability information; a set of link types describing different relations; and traceability processes to ensure traceability and system consistency. The Metamodel contemplates the seven actions defined in the Reference Model plus the action of Activation.

Both models were used as evaluation tools: the Reference Model was used to evaluate two relevant contributions in the literature; the Metamodel was used to evaluate a set of requirements of a course management system. In the former, we have gained knowledge about common flaws of traceability models; in the latter, we have gained knowledge on the usefulness of the Metamodel, and how partial link types and conflicts relate to each

other. By using the models, we intended to exemplify how to use them and also provide a basic proof of concept.

We have also proposed new concepts and definitions for traceability, such as traceability consistency and comprehensiveness of types; these are provided throughout this text. We have also discussed topics we could not find in the literature, such as modeling relations by using one or two link types.

11.1 Reviewing the Contribution

In Chapter 2, we describe a literature review focused on traceability. We did not aim for the review to be all-encompassing; i.e., finding all existing works in traceability. It had two goals: to provide sufficient understanding on the topic, and to enable us to discover existing issues for which we could propose solutions. A set of 6633 papers was discovered and evaluated; a selection of 212 papers was considered of direct interest. A deeper inspection led to selecting a set of 22 papers that we deemed to be the most closely aligned to our research interest.

As a result of the review, we have found that most model-based papers: (i) lack well-defined traceability link types, making the model difficult or even impossible to use; (ii) provide incomplete coverage of situations by not modeling the most common relations between elements; (iii) do not provide mechanisms to ensure traceability and system consistency after changes; (iv) consider only requirements traceability, ignoring other activities of the development process; and (v) lack new concepts and properties for traceability.

In Chapter 3, we propose a Reference Model for traceability; it aims to help with the issues identified in the literature review by defining: the basic elements in a traceability model, how they interact with each other, and the necessary properties to enable the practical use of a model in a software development project. Given these basic elements, it defines basic sets of actions, link types, artifact types, and processes; each set contains the minimum necessary elements for general use. The set of actions defines the minimum set of actions which should be considered by a model. The set of link types defines the minimum set of relations which should be considered by a model. The set of artifact types contains a set of traceable elements which are common in development processes. The set of processes is the minimum set of mechanisms – which a model should have – to ensure consistency in a project. Finally, necessary properties for the sets of link types and artifact types are also defined.

In Chapter 4, we use the Reference Model to evaluate two relevant contributions in model-based traceability: an empirical paper from Ramesh and Jarke [53] and a set of papers by Goknil et al. [23, 21, 22, 24, 25]. Our evaluation focuses on the basic elements in a traceability model: link types, artifact types, and processes. The link types and artifact types of the contribution are mapped to the basic types defined in our Reference Model and evaluated on: level of description and the how they fulfill the three necessary properties. The processes are evaluated on coverage of basic actions and capacity to ensure consistency. We also discuss strengths and/or limitations of each contribution.

In Chapters 5–9, we propose a Metamodel for traceability constructed on top of the Reference Model proposed in Chapter 3. By using the Reference Model as a basis, we hope to provide a model which does not have the issues found in the literature review. The Metamodel also expands on what was defined in the Reference Model by providing: a conceptual model and abstractions of its elements, more specific link types, a new action, a complementary process for one of the basic processes, formalizations of the link types built on the elements defined in the conceptual model, and new artifact types which are useful to record valuable traceability information.

The conceptual model is defined in detail, where each abstraction is described to its smallest elements. This enabled the formal definition of the actions, the artifact types, and the link types. The general link types of the Reference Model are expanded to specific types. Also, a new subtype is proposed: the partial link type; this subtype is useful for situations where multiple artifacts provide the same information. The Activation action is a new action in the Metamodel, being useful to bring inactive artifacts to the Active System. The process for assessing the homologation of two types of Rationales is added to the set of processes of the Metamodel; this process is essential for change impact analysis and to avoid the introduction of inconsistencies in the System Space. Each link type of the set of link types is described and formalized given the basis provided by the conceptual model; this is done to help avoiding ambiguities when mapping the link types to relations between elements of a project. The new artifact types – the Rationale types – are provided to enable recording traceability information such as rationale and description of actions.

The Metamodel is described in the following chapters: in Chapter 5, an outline of the Metamodel is provided. In Chapter 6, the conceptual model is described and defined; the actions contemplated by the Metamodel are also defined here. In Chapter 7, seven new artifact types are defined. In Chapter 8, all fifty-nine link types of the Metamodel, realizing the general link types of the Reference Model, are described in detail. Finally, in Chapter 9, eight processes, realizing the basic processes of the Reference Model, are described as algorithms and as detailed textual descriptions.

In Chapter 10, we use the Metamodel to identify issues with a set of requirements for a course management system. This chapter aims to illustrate the use of the Metamodel and exemplify how to use the processes. However, this example is not robust enough to detail every single aspect of the Metamodel; a working project having used the Metamodel from the beginning, containing a specific state of development (having active and inactive artifacts), actors which interacted with the project, and predefined rules, would be necessary to demonstrate all details of our contribution. Still, we hope this illustration of how to use the Metamodel helps in its understanding.

In conclusion, we hope this contribution helps to strengthen common practice of traceability; the Reference Model aims to provide guidelines to create, or evaluate, traceability models by providing concepts, defining basic elements, and describing how these elements relate to each other. The Metamodel works as a proof of concept of the Reference Model and aims to provide a useful traceability model for software development projects.

11.2 Simplifying the Metamodel

The Metamodel may be simplified according to the specific needs of a project. Some projects may need a leaner approach, choosing to change, or not to use, certain elements of the Metamodel.

Rationale artifact types may be simplified according to necessity. For instance, a Rationale for Modification can be changed to describe only a modification, removing the justification of the action; however, this simplification results in traceability information loss and, by describing only the modification, the Rationale artifact loses the property that makes it a Rationale type.

Simplifying an element may result in customizations of other related elements; e.g., if the Rationale for Modification is changed to describe only the modification, it is necessary to adapt the definitions of a few related link types accordingly.

Link types may be discarded according to the needs of a project. Discarding certain elements may also cause the need to discard related link types. For instance, a project may choose not to use Rationale artifacts and, consequently, it will not use link types which connect rationales to other elements; or a project may choose not to allow the decomposition of artifacts and, therefore, it will not use decomposition-related link types.

Simplification of the link types may incur in traceability information loss; hence, decisions should take into account the traceability cost of removing certain link types.

Like the previous elements, processes may be customized as needed. A few examples: a project may avoid homologating newly created artifacts; a project may use the Homologation Process but not the assessment of RMs and RRs; a project may not use Rationale types, modifying the processes accordingly; a project may not keep previous versions of modified artifacts by customizing the modification process.

There are simplifications which do not truly reduce effort; there is information which will still exist, and be necessary, even if the decision to simplify the Metamodel is taken. For instance, if a project does not uses Rationale types and a decomposition occurs, it loses the capacity to record the justification for the decomposition and the description of how the decomposition should be made; however, the actors in charge of the decomposition still need to know how the artifact should be broken into multiple artifacts. Hence, removing the Rationale for Decomposition may not really ease the effort since the information it records is still needed for the action to be taken. The information will subsequently be lost, but it will exist while the action is being performed. Succintly, the information exists, and is needed, independently of using this artifact type; thus, there is only loss of traceability information by choosing not to use it.

Other customizations may simplify the Metamodel at the expense of added risks. For instance, if a project does not use the Homologation Process and the corresponding rationale type, there is a higher risk of generating problems such as broken dependencies or conflicts. Moreover, the justification for adding an artifact to the system (homologating a non-rationale type), or the justification to perform an action (homologating a rationale type), will also not be recorded. This lost information may be essential to avoid related issues in the future.

On the other hand, there are certain aspects of the Metamodel which may not be possible or desirable for every project. For instance, the Homologation Process requires another actor(s) checking for correctness, necessity, impact on other artifacts, generation of consistency issues, and so on. This kind of verification by another group of actors is not always possible or desirable for smaller projects; thus, each project should decide which parts of the Metamodel are useful for them.

In conclusion, each project should assess which loss of information, and added risks, are acceptable when simplifying the Metamodel. Some simplifications do not really ease the effort of using the Metamodel, some simplifications incur loss of traceability information and/or added risks as consequences, and some simplifications are needed for specific projects.

11.3 Future Work and Limitations

Improving the Properties of Link Types and Artifact Types. There are limitations in the proposed work. The properties of link types and artifact types are well defined but lack formalism; also, there is substantial subjectivity when identifying if a set lacks, and how much it lacks, a property. A more formal definition and more detailed, and rigorous, rules to perform evaluations by using the properties are needed. This should be investigated in the future.

Using the Metamotel in a Project in the Industry. The application of the Metamodel on requirements of a course management system serves as basic proof of concept; however, a more thorough application of the Metamodel – by using it in a working project from the industry – would be useful to identify weaknesses, missing link types, missing concepts and elements in the Traceability Space, etc. It would be easier to use the Metamodel from the beginning of a project; depending on the size of a project, inserting the Metamodel in a working project could cause a considerable amount of effort, since all existing relations, artifacts, and if they are active or inactive, would have to be identified before any changes are done, or changes would have to be recorded in detail to update the Traceability Space accordingly.

Proposing Metrics Taking Into Account the Traceability Space. There are several aspects which may be improved and developed in our proposal. Metrics could be created by taking into account information which may be inferred from the Traceability Space of a project; a few starting suggestions are provided in Section 6.8 in Chapter 6, such as: the identification of overworked actors by comparing each actor number of Accountability links, or identification of relevant Artifacts in a project by counting the number of NecessaryFor links starting in it; this also enables the possibility of selecting this Artifact for decomposition.

Developing Supporting Tools. Tools could be developed to support the Reference Model and the Metamodel. A tool to support the Reference Model could help in the

evaluation and creation of traceability models; a tool to support the Metamodel could provide: semi-automated use of the processes, recording of all traceability links between artifacts and actors, change impact analysis, among other functionalities. A simple tool for the Metamodel would record the artifacts, the links between them, and identify a few common inconsistencies; a more useful tool would provide the steps necessary to ensure consistency given a change, automatically update traceability links, rate the cost of a change taking into account each different recommended path to keep the project consistent, and identify inconsistencies in a working project.

Extending the Sets of Link Types and Artifact Types. The sets of link types and artifact types of the Metamodel could be extended to cover more domains, such as safety-focused projects, or to increase its comprehensiveness and specificity; e.g., the dependency relation could be modeled by more link types, representing different types of dependency between artifacts. Currently, the dependency relation is not specified; however, there may be several types of dependencies which could be identified, such as existential dependency and dependency on the previous application of an artifact.

Investigating the Graph Structure of the System Space. New angles could be explored given the graph structure of the System Space. Paths, or trees, built by traceability links could be studied for change impact analysis; for instance, do homogeneous paths – same link type, or similar link type – and heterogeneous paths – different link type – have different properties concerning impact of changes? That is, are there properties which may be assessed from a System Space graph that could be useful for change impact analysis? Are there properties which may be assessed given other types of analysis?

Adding New Concepts. Finally, new concepts and elements to enrich traceability focusing in reducing subjectivity could be investigated; traceability, while being a topic of research for over 40 years, still does not have a strong conceptual basis.

Bibliography

[1] F. Abbors, A. Bäcklund, and D. Truscan. Matera - an integrated framework for model-based testing. In *2010 17th IEEE International Conference and Workshops on Engineering of Computer Based Systems*, pages 321–328, March 2010.

[2] N. Aizenbud-Reshef, B. T. Nolan, J. Rubin, and Y. Shaham-Gafni. Model traceability. *IBM Systems Journal*, 45(3):515–526, 2006.

[3] G. Alaa and Z. Samir. A multi-faceted roadmap of requirements traceability types adoption in scrum: An empirical study. In *2014 9th International Conference on Informatics and Systems*, pages SW–1–SW–9, Dec 2014.

[4] Joachim Bayer and Tanya Widen. Introducing traceability to product lines. In Frank van der Linden, editor, *Software Product-Family Engineering*, pages 409–416, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[5] B. Berenbach and Timo Wolf. A unified requirements model; integrating features, use cases, requirements, requirements analysis and hazard analysis. In *Global Software Engineering, 2007. ICGSE 2007. Second IEEE International Conference on*, pages 197–203, Aug 2007.

[6] A. F. Binti Arbain, I. Ghani, and W. M. N. Wan Kadir. Agile non functional requiremnents (nfr) traceability metamodel. In *2014 8th. Malaysian Software Engineering Conference (MySEC)*, pages 228–233, Sep. 2014.

[7] B. W. Boehm. Software engineering. *IEEE Trans. Comput.*, 25(12):1226–1241, December 1976.

[8] Rafael Capilla, Olaf Zimmermann, Uwe Zdun, Paris Avgeriou, and Jochen M. Küster. An enhanced architectural knowledge metamodel linking architectural design decisions to other artifacts in the software engineering lifecycle. In Ivica Crnkovic, Volker Gruhn, and Matthias Book, editors, *Software Architecture*, pages 303–318, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[9] J. Cleland-Huang, G. Zemont, and W. Lukasik. A heterogeneous solution for improving the return on investment of requirements traceability. In *Requirements Engineering Conference, 2004. Proceedings. 12th IEEE International*, pages 230–239, Sept 2004.

[10] Jane Cleland-Huang, Carl K. Chang, and Jeffrey C. Wise. Automating performance-related impact analysis through event based traceability. *Requir. Eng.*, 8(3):171–182, 2003.

[11] Jane Cleland-Huang, Orlena Gotel, and Andrea Zisman. *Software and Systems Traceability*. Springer Publishing Company, Incorporated, 2012.

[12] Jane Cleland-Huang and David Schmelzer. Dynamically tracing non-functional requirements through design pattern invariants. *Workshop on Traceability in Emerging Forms of Software Engineering, in conjunction with IEEE International Conference on Automated Software Engineering*, 2003.

[13] Diego Dermeval, Jaelson Castro, Carla Silva, João Pimentel, Ig Ibert Bittencourt, Patrick Brito, Endhe Elias, Thyago Tenório, and Alan Pedro. On the use of metamodeling for relating requirements and architectural design decisions. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, pages 1278–1283, New York, NY, USA, 2013. ACM.

[14] Jessica Díaz, Jennifer Pérez, Juan Garbajosa, and Alexander L. Wolf. Change impact analysis in product-line architectures. In Ivica Crnkovic, Volker Gruhn, and Matthias Book, editors, *Software Architecture*, pages 114–129, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[15] A. J. J. Dick. Evidence-based development - coupling structured argumentation with requirements development. In *7th IET International Conference on System Safety, incorporating the Cyber Security Conference 2012*, pages 1–5, Oct 2012.

[16] H. Dubois, M. A. Peraldi-Frati, and F. Lakhal. A model for requirements traceability in a heterogeneous model-based design process: Application to automotive embedded systems. In *2010 15th IEEE International Conference on Engineering of Complex Computer Systems*, pages 233–242, March 2010.

[17] A. P. . Eberlein, M. J. Crowther, and F. Halsall. Rats: a software tool to aid the transition from service idea to service implementation. In *Proceedings of GLOBECOM'96. 1996 IEEE Global Telecommunications Conference*, volume 3, pages 1991–1995 vol.3, Nov 1996.

[18] Angelina Espinoza, Goetz Botterweck, and Juan Garbajosa. A formal approach to reuse successful traceability practices in spl projects. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 2352–2359, New York, NY, USA, 2010. ACM.

[19] I Galvao and A Goknil. Survey of traceability approaches in model-driven engineering. In *Enterprise Distributed Object Computing Conference, 2007. EDOC 2007. 11th IEEE International*, pages 313–313, Oct 2007.

[20] H. E. ghazi. Mv - tmm: A multi view traceability management method. In *2008 32nd Annual IEEE International Computer Software and Applications Conference*, pages 247–254, July 2008.

[21] Arda Goknil, Ivan Kurtev, Klaas Berg, and Jan-Willem Veldhuis. Semantics of trace relations in requirements models for consistency checking and inferencing. *Software and Systems Modeling (SoSyM)*, 10(1):31–54, February 2011.

[22] Arda Goknil, Ivan Kurtev, and Jean-Vivien Millo. A metamodeling approach for reasoning on multiple requirements models. In *Proceedings of the 2013 17th IEEE International Enterprise Distributed Object Computing Conference*, EDOC '13, pages 159–166, Washington, DC, USA, 2013. IEEE Computer Society.

[23] Arda Goknil, Ivan Kurtev, and Klaas van den Berg. *A Metamodeling Approach for Reasoning about Requirements*, pages 310–325. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[24] Arda Goknil, Ivan Kurtev, and Klaas Van Den Berg. Generation and validation of traces between requirements and architecture based on formal trace semantics. *Journal of Systems and Software*, 88:112–137, February 2014.

[25] Arda Goknil, Ivan Kurtev, Klaas van den Berg, and Wietze Spijkerman. Change impact analysis for requirements: A metamodeling approach. *Information & Software Technology*, 56(8):950–972, 2014.

[26] O. C Z Gotel and A C W Finkelstein. An analysis of the requirements traceability problem. In *Requirements Engineering, 1994., Proceedings of the First International Conference on*, pages 94–101, Apr 1994.

[27] Sol J. Greenspan and Clement L. McGowan. Structuring software development for reliability. *Microelectronics Reliability*, 17(1):75 – 83, 1978.

[28] S. Haidrar, A. Anwar, and O. Roudies. Towards a generic framework for requirements traceability management for sysml language. In *2016 4th IEEE International Colloquium on Information Science and Technology (CiSt)*, pages 210–215, Oct 2016.

[29] J. Han. Tram: a tool for requirements and architecture management. In *Proceedings 24th Australian Computer Science Conference. ACSC 2001*, pages 60–68, Jan 2001.

[30] D. Hetherinton. Sysml requirements for training game design. In *17th International IEEE Conference on Intelligent Transportation Systems (ITSC)*, pages 162–167, Oct 2014.

[31] E. Horowitz and R. C. Williamson. Sodos: A software documentation support environment - its use. *IEEE Transactions on Software Engineering*, SE-12(11):1076–1087, Nov 1986.

[32] E. Horqwitz and R. C. Williamson. Sodos: A software documentation support environment - its definition. *IEEE Transactions on Software Engineering*, SE-12(8):849–859, Aug 1986.

[33] Y. Hu and B. Panda. Two-dimensional traceability link rule mining for detection of insider attacks. In *2010 43rd Hawaii International Conference on System Sciences*, pages 1–9, Jan 2010.

[34] IEEE. 830-1998 - IEEE recommended practice for software requirements specifications. Acessed February 21 2015.

[35] W. Jirapanthong and A. Zisman. Supporting product line development through traceability. In *12th Asia-Pacific Software Engineering Conference (APSEC'05)*, pages 9 pp.–, Dec 2005.

[36] M. Kassab, O. Ormandjieva, and M. Daneva. A traceability metamodel for change management of non-functional requirements. In *2008 Sixth International Conference on Software Engineering Research, Management and Applications*, pages 245–254, Aug 2008.

[37] M. Kassab, O. Ormandjieva, and M. Daneva. A metamodel for tracing non-functional requirements. In *2009 WRI World Congress on Computer Science and Information Engineering*, volume 7, pages 687–694, March 2009.

[38] Gerald Kotonya and Ian Sommerville. *Requirements Engineering: Processes and Techniques*. Wiley Publishing, 1st edition, 1998.

[39] Patricio Letelier, Elena Navarro, and Víctor Anaya. Customizing traceability in a software development process. In Olegas Vasilecas, Wita Wojtkowski, Jože Zupančič, Albertas Caplinskas, W. Gregory Wojtkowski, and Stanisław Wrycza, editors, *Information Systems Development*, pages 137–148, Boston, MA, 2005. Springer US.

[40] Mikael Lindvall and Kristian Sandahl. Practical implications of traceability. *Softw. Pract. Exper.*, 26(10):1161–1180, October 1996.

[41] P. Mäder, O. Gotel, and I Philippow. Getting back to basics: Promoting the use of a traceability information model in practice. In *Traceability in Emerging Forms of Software Engineering, 2009. TEFSE '09. ICSE Workshop on*, pages 21–25, May 2009.

[42] Patrick Mäder, Ilka Philippow, and Matthias Riebisch. Customizing traceability links for the unified process. In Sven Overhage, Clemens A. Szyperski, Ralf Reussner, and Judith A. Stafford, editors, *Software Architectures, Components, and Applications*, pages 53–71, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[43] P. Maeder, I. Philippow, and M. Riebisch. A traceability link model for the unified process. In *Eighth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD 2007)*, volume 3, pages 700–705, July 2007.

[44] Ivano Malavolta, Henry Muccini, and V. Smrithi Rekha. Supporting architectural design decisions evolution through model driven engineering. In Elena A. Troubitsyna,

editor, *Software Engineering for Resilient Systems*, pages 63–77, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[45] Kannan Mohan, Peng Xu, Lan Cao, and Balasubramaniam Ramesh. Improving change management in software development: Integrating traceability and software configuration management. *Decision Support Systems*, 45(4):922 – 936, 2008. Information Technology and Systems in the Internet-Era.

[46] Sunil Nair, Jose Luis de la Vara, Alberto Melzi, Giorgio Tagliaferri, Laurent de-la Beaujardiere, and Fabien Belmonte. Safety evidence traceability: Problem analysis and model. In Camille Salinesi and Inge van de Weerd, editors, *Requirements Engineering: Foundation for Software Quality*, pages 309–324, Cham, 2014. Springer International Publishing.

[47] Shiva Nejati, Mehrdad Sabetzadeh, Davide Falessi, Lionel Briand, and Thierry Coq. A sysml-based approach to traceability management and design slicing in support of safety certification: Framework, tool support, and case studies. *Information and Software Technology*, 54(6):569 – 590, 2012.

[48] Richard F. Paige, Gøran K. Olsen, Dimitrios S. Kolovos, Steffen Zschaler, and Christopher Power. Building model-driven engineering traceability classifications. In *ECMDA Traceability Workshop (ECMDA-TW) 2008 Proceedings*, pages 49–58, 2008.

[49] R. K. Panesar-Walawege, M. Sabetzadeh, L. Briand, and T. Coq. Characterizing the chain of evidence for software safety cases: A conceptual model based on the iec 61508 standard. In *2010 Third International Conference on Software Testing, Verification and Validation*, pages 335–344, April 2010.

[50] S. L. Pfleeger and S. A. Bohner. A framework for software maintenance metrics. In *Proceedings. Conference on Software Maintenance 1990*, pages 320–327, Nov 1990.

[51] B. Ramesh, D. Dwiggins, G. DeVries, and M. Edwards. Towards requirements traceability models. In *Systems Engineering of Computer Based Systems, 1995., Proceedings of the 1995 International Symposium and Workshop on*, pages 229–232, 1995.

[52] B. Ramesh and M. Edwards. Issues in the development of a requirements traceability model. In *Requirements Engineering, 1993., Proceedings of IEEE International Symposium on*, pages 256–259, Jan 1993.

[53] B. Ramesh and M. Jarke. Toward reference models for requirements traceability. *Software Engineering, IEEE Transactions on*, 27(1):58–93, Jan 2001.

[54] B. Ramesh, T. Powers, C. Stubbs, and M. Edwards. Implementing requirements traceability: a case study. In *Requirements Engineering, 1995., Proceedings of the Second IEEE International Symposium on*, pages 89–95, Mar 1995.

[55] Brian Randell. Towards a methodology of computing system design. *NATO Software Engineering Conference*, pages 204–208, October 1968.

[56] J. Richardson and J. Green. Automating traceability for generated software artifacts. In *Automated Software Engineering, 2004. Proceedings. 19th International Conference on*, pages 24–33, Sept 2004.

[57] S. Rochimah, W.M.N.W. Kadir, and AH. Abdullah. An evaluation of traceability approaches to support software evolution. In *Software Engineering Advances, 2007. ICSEA 2007. International Conference on*, pages 19–19, Aug 2007.

[58] P. Sanchez, D. Alonso, F. Rosique, B. Alvarez, and J. A. Pastor. Introducing safety requirements traceability support in model-driven development of robotic applications. *IEEE Transactions on Computers*, 60(8):1059–1071, Aug 2011.

[59] Maurício Serrano and Julio Cesar Sampaio do Prado Leite. A rich traceability model for social interactions. In *Proceedings of the 6th International Workshop on Traceability in Emerging Forms of Software Engineering*, TEFSE '11, pages 63–66, New York, NY, USA, 2011. ACM.

[60] L. Shen, X. Peng, and W. Zhao. A comprehensive feature-oriented traceability model for software product line development. In *2009 Australian Software Engineering Conference*, pages 210–219, April 2009.

[61] Scott Sigman and Xiaoqing Frank Liu. A computational argumentation methodology for capturing and analyzing design rationale arising from multiple perspectives. *Information and Software Technology*, 45(3):113 – 122, 2003.

[62] IEEE Computer Society, Pierre Bourque, and Richard E. Fairley. *Guide to the Software Engineering Body of Knowledge (SWEBOK(R)): Version 3.0.* IEEE Computer Society Press, Los Alamitos, CA, USA, 3rd edition, 2014.

[63] A. Tang and J. Han. Architecture rationalization: a methodology for architecture verifiability, traceability and completeness. In *12th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'05)*, pages 135–144, April 2005.

[64] Antony Tang, Yan Jin, and Jun Han. A rationale-based architecture model for design traceability and reasoning. *Journal of Systems and Software*, 80(6):918 – 934, 2007.

[65] Masoumeh Taromirad and Richard F. Paige. Agile requirements traceability using domain-specific modelling languages. In *Proceedings of the 2012 Extreme Modeling Workshop*, XM '12, pages 45–50, New York, NY, USA, 2012. ACM.

[66] Bedir Tekinerdoğan, Christian Hofmann, Mehmet Akşit, and Jethro Bakker. *Metamodel for Tracing Concerns Across the Life Cycle.* Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[67] Huy Tran, Uwe Zdun, and Schahram Dustdar. Vbtrace: using view-based and model-driven development to support traceability in process-driven soas. *Software & Systems Modeling*, 10(1):5–29, Feb 2011.

[68] Jan-Willem Veldhuis. Tool for requirements inferencing and consistency checking (tric). Accessed May 8 2018.

[69] Rainer Weinreich and Georg Buchgeher. Integrating requirements and design decisions in architecture representation. In Muhammad Ali Babar and Ian Gorton, editors, *Software Architecture*, pages 86–101, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[70] Stefan Winkler and Jens Pilgrim. A survey of traceability in requirements engineering and model-driven development. *Softw. Syst. Model.*, 9(4):529–565, September 2010.

[71] Wei Zhang, Hong Mei, and Haiyan Zhao. Feature-driven requirement dependency analysis and high-level software design. *Requirements Engineering*, 11(3):205–220, Jun 2006.

[72] L. Zhu and I. Gorton. Uml profiles for design decisions and non-functional requirements. In *Second Workshop on Sharing and Reusing Architectural Knowledge - Architecture, Rationale, and Design Intent (SHARK/ADI'07: ICSE Workshops 2007)*, pages 8–8, May 2007.

Appendix A

List of Papers of the Initial Selection

Table A.1: Papers Obtained in the Initial Selection – Springer Link

Name	Year
Automatic Support for Traceability in a Generic Model Management Framework	2005
An Adaptable ORM Metamodel to Support Traceability of Business Requirements across System Development Life Cycle Phases	2008
Metamodel for Tracing Concerns Across the Life Cycle	2007
Development of a Metamodel to Foster Interoperability along the Product Lifecycle Traceability	2006
Rigorous identification and encoding of trace-links in model-driven engineering	2011
Engineering a DSL for Software Traceability	2009
Traceability Links in Model Transformations between Software and Performance Models	2013
NFR+ framework method to support bi-directional traceability of non-functional requirements	2012
An Enhanced Architectural Knowledge Metamodel Linking Architectural Design Decisions to other Artifacts in the Software Engineering Lifecycle	2011
A model for tracing variability from features to product-line architectures: a case study in smart grids	2014
A Metamodeling Approach for Reasoning about Requirements	2005
Managing requirements uncertainty with partial models	2013
Safety Evidence Traceability: Problem Analysis and Model	2014
Customizing Traceability Links for the Unified Process	2007
A Trace Management Platform for Risk-Based Security Testing	2014
Introducing Traceability to Product Lines	2002
Incorporating Traceability in Conceptual Models for Data Warehouses by Using MDA	2011
Automating the Trace of Architectural Design Decisions and Rationales Using a MDD Approach	2008
A Metamodeling Approach for Reasoning about Requirements	2008
Change Impact Analysis in Product-Line Architectures	2011
Enterprise Modeling and Decision-Support for Automating the Business Rules Lifecycle	2002
A Visual Traceability Modeling Language	2010
An Automated Approach to Transform Use Cases into Activity Diagrams	2010
Integrating ontologies, model driven, and CNL in a multi-viewed approach for requirements engineering	2011
Managing Data Warehouse Traceability: A Life-Cycle Driven Approach	2015
Requirements Traceability in Agent Oriented Development	2003
Using Rules for Traceability Creation	2012
Aspects at the Right Time	2007
Continued on next page	

Supporting Architectural Design Decisions Evolution through Model Driven Engineering	2011
A Meta-model for Requirements Engineering in System Family Context for Software Process Improvement Using CMMI	2005
A Pattern-Based Approach towards the Guided Reuse of Safety Mechanisms in the Automotive Domain	2014
Requirements-Driven Software Service Evolution	2013
The Change Impact Analysis in BPM Based Software Applications: A Graph Rewriting and Ontology Based Approach	2014
DO-333 Certification Case Studies	2014
Requirements Management: The HOOD Capability Model for Requirements Management (Book Chapter)	2008
Feature-driven requirement dependency analysis and high-level software design	2006
Integrating Requirements and Design Decisions in Architecture Representation	2010
The Decision View of Software Architecture	2005
Adaptive socio-technical systems: a requirements-based approach	2013
An intelligent assistant for requirements validation	1995
KBRE: a framework for knowledge-based requirements engineering	2014
Methods for Creating Co-models of Embedded Systems	2014
The PLUSS Approach – Domain Modeling with Features, Use Cases and Use Case Realizations	2005
Software Quality Engineering: The Leverage for Gaining Maturity	2008
Requirements-Based Estimation of Change Costs	2000
Tracking Design Dependencies to Support Conflict Management	2007
Industry evaluation of the Requirements Abstraction Model	2007
A Knowledge-Based and Model-Driven Requirements Engineering Approach to Conceptual Satellite Design	2009
Combining Architectural Design Decisions and Legacy System Evolution	2014
Requirements Engineering: Advanced Traceability (Book Chapter)	2011
Tracing Non-Functional Requirements	2011
A System Requirements Traceability Model: An Industrial Application	1998
Understanding Model Transformations	2015
Application of an Extended SysML Requirements Diagram to Model Real-Time Control Systems	2013
User Driven Evolution of User Interface Models – The FLEPR Approach	2011
Macro-level Traceability Via Media Transformations	2008
Development of a Software Tool to Support Traceability-Based Inspection of SOFL Specifications	2015
VbTrace: using view-based and model-driven development to support traceability in process-driven SOAs	2011
SCADE: A Comprehensive Framework for Critical System and Software Engineering	2012
MDI: A Rule-based Multi-document and Tool Integration Approach	2006
Application of Quality Standards to Multiple Artifacts with a Universal Compliance Solution	2010
Towards a Practical Approach to Check UML/fUML Models Consistency Using CSP	2011
A Deductive View on Process-Data Diagrams	2011
Supporting fine-grained traceability in software development environments	2006
Requirements Traceability across Organizational Boundaries - A Survey and Taxonomy	2013
A Survey on Usage Scenarios for Requirements Traceability in Practice	2013
An Ontology for Quality Management – Enabling Quality Problem Identification and Tracing	1999
Model-Based Requirements Engineering for Product Lines	2000
A Metamodeling Approach for Reasoning about Requirements	2008
Semantics of trace relations in requirements models for consistency checking and inferencing	2011

Table A.2: Papers Obtained in the Initial Selection – ScienceDirect

Name	Year
Transforming and tracing reused requirements models to home automation models	2013
A method and tool for tracing requirements into specifications	2014
Autonomic tracing of production processes with mobile and agent-based computing	2011
Integrating visual goal models into the Rational Unified Process	2006
Model-Driven Engineering as a new landscape for traceability management: A systematic literature review	2012
A SysML-based approach to traceability management and design slicing in support of safety certification: Framework, tool support, and case studies	2012
Towards automated traceability maintenance	2012
Improving reviews of conceptual models by extended traceability to captured system usage	2000
A computational argumentation methodology for capturing and analyzing design rationale arising from multiple perspectives	2003
System Requirements Analysis: Requirements Traceability Relationships (Book Chapter)	2006
A document driven methodology for developing a high quality Parallel Mesh Generation Toolbox	2009
A trace metamodel proposal based on the model driven architecture framework for the traceability of user requirements in data warehouses	2012
Introducing requirements traceability support in model-driven development of web applications	2009
Linking requirements and design data for automated functional evaluation	1996
Structuring software development for reliability	1978
Medical device standards' requirements for traceability during the software development lifecycle and implementation of a traceability assessment model	2013
Software Patterns for Traceability of Requirements to Finite State Machine Behavior	2012
Automated traceability analysis for UML model refinements	2009
Specifying and building interoperable eHealth systems: ODP benefits and lessons learned	2013
DRAMA: A framework for domain requirements analysis and modeling architectures in software product lines	2008
A scoped approach to traceability management	2009
Improving change management in software development: Integrating traceability and software configuration management	2008
A rationale-based architecture model for design traceability and reasoning	2007
Rule-based generation of requirements traceability relations	2004
Traceability-centric model-driven object-oriented engineering	2010
Generation and validation of traces between requirements and architecture based on formal trace semantics	2014
Change impact analysis for requirements: A metamodeling approach	2014

Table A.3: Papers Obtained in the Initial Selection – IEEE Xplore Digital Library

Name	Year
A Proposal for Defining a Set of Basic Items for Project-Specific Traceability Methodologies	2008
SOMA-ME: A platform for the model-driven design of SOA solutions	2008
A model-driven visualization tool for use with Model-Based Systems Engineering projects	2014
A Model for Requirements Traceability in a Heterogeneous Model-Based Design Process: Application to Automotive Embedded Systems	2010
Validation of data warehouse requirements - model traceability metrics using a formal framework	2015
Towards an integrated systems engineering environment	2004
Continued on next page	

Traceability in digital forensic investigation process	2011
Incorporating Multimedia Source Materials into a Traceability Framework	2006
Hidden Implementation Dependencies in High Assurance and Critical Computing Systems	2006
A framework for software maintenance metrics	1990
Integration of Requirements Engineering and Test-Case Generation via OSLC	2014
Migration from Procedural Programming to Aspect Oriented Paradigm	2009
A structured goal based measurement framework enabling traceability and prioritization	2010
A metamodel for tracing requirements of real-time systems	2013
Model-based protocol engineering: Specifying Kerberos with object-process methodology	2014
Formalizing "Traceability' [sic] for Architectural Evolutions	2010
Traceability management framework for patient data in healthcare environment	2010
Executable architecture modeling and validation	2010
Meta-modelling approach to traceability for avionics: a framework for managing the engineering of computer based aerospace systems	2003
Toward reference models for requirements traceability	2001
Model-based traceability	2009
An Ontology-Based Approach for Multiperspective Requirements Traceability between Analysis Models	2010
A tactic-centric approach for automating traceability of quality concerns	2012
A Traceability Link Model for the Unified Process	2007
Towards requirements traceability models	1995
Softgoal Traceability Patterns	2006
An approach to carry out consistency analysis on requirements: Validating and tracking requirements through a configuration structure	2013
Extension Features-Driven Use Case Model for requirement traceability	2009
A Study on the Effect of Traceability Links in Software Maintenance	2013
A Comprehensive Feature-Oriented Traceability Model for Software Product Line Development	2009
Two-Dimensional Traceability Link Rule Mining for Detection of Insider Attacks	2010
Analyzing and Systematizing Current Traceability Schemas	2006
Introducing Safety Requirements Traceability Support in Model-Driven Development of Robotic Applications	2011
Examining Communication Media Selection and Information Processing in Software Development Traceability: An Empirical Investigation	2009
A means of establishing traceability based on a UML model in business application development	2011
Getting back to basics: Promoting the use of a traceability information model in practice	2009
A multi-faceted roadmap of requirements traceability types adoption in SCRUM: An empirical study	2014
A unified requirements model; integrating features, use cases, requirements, requirements analysis and hazard analysis	2007
Requirement traceability: A model-based approach	2014
MV - TMM: A Multi View Traceability Management Method	2008
Implementing requirements traceability: a case study	1995
Ontology-based model for Rational Unified Process artifacts traceability	2012
Supporting product line development through traceability	2005
A software model for impact analysis: a validation experiment	1999
An Integrated Decision Model For Efficient Requirement Traceability In SPICE Compliant Development	2007
A multi view based traceability management method	2008
Continued on next page	

A decision model for managing and communicating resource restrictions in embedded systems design	2008
Relational-model based change management for non-functional requirements: Approach and experiment	2011
Traceability between Software Architecture Models	2006
Integrating UML, MARTE and SysML to improve requirements specification and traceability in the embedded domain	2014
An exploratory case study of the maintenance effectiveness of traceability models	2000
Issues in the development of a requirements traceability model	1993
Characterizing the Chain of Evidence for Software Safety Cases: A Conceptual Model Based on the IEC 61508 Standard	2010
Improving Software Quality through Requirements Traceability Models	2006
A Tool-Based Methodology for System Testing of Service-Oriented Systems	2010
Requirements engineering in a model-based methodology for embedded automotive software	2008
Managing requirements uncertainty with partial models	2012
Dynamic traceability links supported by a system architecture description	1997
Architecture rationalization: a methodology for architecture verifiability, traceability and completeness	2005
Towards a unified Requirements Modeling Language	2010
Experiences from a model-based methodology for embedded electronic software in automobile	2008
Agile non functional requiremnents [sic] (NFR) traceability metamodel	2014
Systematic requirements recycling through abstraction and traceability	2002
Supporting evolutionary development by feature models and traceability links	2004
Architecting for evolvability by means of traceability and features	2008
Modeling and design of service-oriented architecture	2004
Using tactic traceability information models to reduce the risk of architectural degradation during system maintenance	2011
Enhancing requirements and change management through process modelling and measurement	2011
SODOS: A software documentation support environment – Its use	1986
Ontology-based user requirements decomposition for component selection for highly available systems	2014
Managing Evolution by Orchestrating Requirements and Testing Engineering Processes	2012
A Feature-Oriented Requirements Tracing Method: A Study of Cost-benefit Analysis	2006
Bridging the gap between past and future in RE: a scenario-based approach	1999
A product data dependencies network to support conflict resolution in design processes	2006
UML Profiles for Design Decisions and Non-Functional Requirements	2007
Evidence-based development - coupling structured argumentation with requirements development	2012
A Traceable Maturity Assessment Method Based on Enterprise Architecture Modelling	2014
Requirements Management Tool with Evolving Traceability for Heterogeneous Artifacts in the Entire Life Cycle	2010
SODOS: A software documentation support environment – Its definition	1986
The evolution support environment system	1990
Agent-based knowledge keep tracking	2003
A Traceability Metamodel for Change Management of Non-functional Requirements	2008
A scenario-driven approach to trace dependency analysis	2003
Breaking the big-bang practice of traceability: Pushing timely trace recommendations to project stakeholders	2012
MATERA - An Integrated Framework for Model-Based Testing	2010
RATS: a software tool to aid the transition from service idea to service implementation	1996
Continued on next page	

TRAM: a tool for requirements and architecture management	2001
SysML requirements for training game design	2014
A Metamodel for Tracing Non-functional Requirements	2009
Formalizing standards and regulations variability in longlife projects. A challenge for Model-driven engineering	2011
Requirements view for enterprise architectures	2017
Towards a generic framework for requirements traceability management for SysML language	2016
A metamodeling approach for reasoning on multiple requirements models	2013
An exploratory case study of the maintenance effectiveness of traceability models	2000

Table A.4: Papers Obtained in the Initial Selection – ACM Digital Library

Name	Year
Mind the gap: assessing the conformance of software traceability to relevant guidelines	2014
A state-based approach to traceability maintenance	2010
A rich traceability model for social interactions	2011
A hierarchical model for traceability between requirements and architecture	2014
Requirements engineering: from craft to discipline	2008
Requirement traceability in safety critical systems	2010
Reconstructing requirements coverage views from design and test using traceability recovery via LSI	2005
A taxonomy for requirements engineering and software test alignment	2014
A reusable traceability framework using patterns	2005
Mining and analysing security goal models in health information systems	2009
On the use of metamodeling for relating requirements and architectural design decisions	2013
Agile requirements traceability using domain-specific modelling languages	2012
Use of Semi-Formal and Formal Methods in Requirement Engineering of ILMS	2015
Analysis of crosscutting features in software product lines	2008
Requirements variability models: meta-model based transformations	2005
A formal approach to reuse successful traceability practices in SPL projects	2010
Traceability and model checking to support safety requirement verification	2014
Transforming trace information in architectural documents into re-usable and effective traceability links	2011
Tool support for generation and validation of traces between requirements and architecture	2010
Automated change impact analysis between SysML models of requirements and design	2016
Tarski: a platform for automated analysis of dynamically configurable traceability semantics	2017