Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Coordenação de Pós-Graduação em Ciência da Computação

Uma Abordagem para Representação e Rastreio de Artefatos

Arthur de Sousa Marques

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Campina Grande - Campus I como parte dos requisitos necessários para obtenção do grau de Mestre em Ciência da Computação.

Área de Concentração: Ciência da Computação
Linha de Pesquisa: Engenharia de Software

Franklin Ramalho e Wilkerson L. Andrade
(Orientadores)

Campina Grande, Paraíba, Brasil

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA CENTRAL DA UFCG

M357a Marques, Arthur de Sousa.
Uma abordagem para representação e rastreio de artefatos / Arthur de Sousa Marques. – Campina Grande, 2014.
99 f. : il. color.

Dissertação (Mestrado em Ciência da Computação) – Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática, 2014.

"Orientação: Prof. Dr. Franklin Ramalho, Prof. Dr. Wilkerson de Lucena Andrade".
Referências.

1. Engenharia de Software. 2. Rastreabilidade de Requisitos. 3. Linguagem de Representação de Rastreabilidade (TRL). I. Ramalho, Franklin. II. Andrade, Wilkerson de Lucena. III. Título.

CDU 004.41(043)

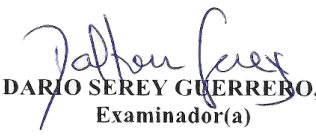
"UMA ABORDAGEM PARA REPRESENTAÇÃO E RASTREIO DE ARTEFATOS"

ARTHUR DE SOUSA MARQUES

DISSERTAÇÃO APROVADA EM 15/12/2014

FRANKLIN DE SOUZA RAMALHO, Dr., UFCG
Orientador(a)

WILKERSON DE LUCENA ANDRADE, D.Sc, UFCG
Orientador(a)



DALTON DARIO SEREY GUERRERO, D.Sc, UFCG
Examinador(a)

ROBERTA DE SOUZA COELHO, Drª, UFRN
Examinador(a)

CAMPINA GRANDE - PB

Resumo

Rastreabilidade de Requisitos refere-se ao processo de rastreio de requisitos ao longo de todo o ciclo de vida de um software. Visto que um grande conjunto de informações é usado e produzido e tais devem ser rastreadas, ela é essencial ao processo de desenvolvimento de software. Não obstante, uma vez que a complexidade dos sistemas desenvolvidos cresce, a miríade de artefatos relacionados também cresce. Sendo assim, engenheiros de requisitos são encarregados de rastrear requisitos em diferentes níveis de abstrações. Neste contexto, vale ressaltar que não há um consenso acerca do processo de rastreabilidade e, como consequência, práticas de rastreabilidade de requisitos não podem ser unificadas em diferentes ambientes organizacionais. Propor uma abstração comum para rastreabilidade de requisitos e também identificar aspectos chave do processo de rastreabilidade são reconhecidos como notáveis tópicos de pesquisa dentre os grandes desafios da rastreabilidade de requisitos. Sendo assim, no presente trabalho, propomos uma Linguagem de Representação de Rastreabilidade (TRL), que provê abstrações para a rastreabilidade de requisitos. Tal linguagem é então explorada por um processo de rastreabilidade, centrado na mesma. Desta forma, ao discutirmos detalhadamente as fases do processo proposto, atores, responsabilidades, entradas e saídas esperadas bem como contratos e interfaces que regem tal processo, nós investigamos aspectos comuns do processo de rastreabilidade. A avaliação do presente trabalho considera que: *(i)* a representação proposta foi avaliada considerando critérios de legibilidade e redigibilidade, ou seja, quão compreensível ela é; e *(ii)* o processo proposto foi avaliado considerando sua *performance* e eficiência, isto é, quão bem o processo apoia atividades beneficiadas pela rastreabilidade de requisitos. Como resultados, observamos que a linguagem e suas construções foram avaliadas como de fácil leitura e escrita e que a linguagem é uma abordagem viável para abstrair rastreabilidade de requisitos. Além disso, observamos que o processo proposto possui melhor *performance* e eficiência quando comparado à um processo *ad hoc*. Dados os resultados observados, a abordagem proposta (linguagem e processo) fornece abstrações para o processo de rastreabilidade de requisitos bem como fomentar a discussão acerca dos principais aspectos do processo de rastreabilidade, desta forma, promovendo a rastreabilidade de requisitos portável.

Abstract

Requirements Traceability (RT) refers to the process of tracing requirements through the software development life-cycle. It is essential for the software development process because a lot of information is used and produced and it should be kept related or traceable. Nevertheless, as the complexity of a system increases, the myriad of related artifacts also increases. Therefore, one is encumbered of tracing requirements through different abstraction levels. Moreover, there is not a consensus about the traceability process and, as a consequence, requirements traceability practices cannot be unified across different organizational settings. Proposing a common abstraction to requirements traceability and also identifying common aspects to the requirements traceability process have been recognized as remarkable research topics of the grand challenges of requirements traceability. Therefore, is this work, we propose a Traceability Representation Language (TRL), which provides abstractions to requirements traceability. Such representation is then exploited by a requirements traceability process centered on it. Thus, by thoroughly discussing process' phases, activities, actors, responsibilities, and input/output artifacts as well as traceability contracts, which govern process' phases and how they intercommunicate, we investigate common aspects of requirements traceability. The evaluation of the present work was twofold: *(i)* the proposed language was evaluated considering its readability and writability, *i.e.* how comprehensible it is; and *(ii)* the proposed process was evaluated regarding its performance and effectiveness, *i.e.* how well it supports requirements traceability tasks. As a result, we observed that the language's constructions were evaluated as easily read/written and that it is a feasible approach to provide an abstraction to requirements traceability. Moreover, we observed that the proposed process improves the performance and efficiency of the requirements traceability process, while maintaining the same accuracy of other approaches. Therefore, the proposed approach (language and process) is feasible to address abstractions to requirements traceability as well as foster the discussion of major aspects of the requirements traceability process, thus portable traceability can be addressed, *i.e.* how requirements traceability techniques can be used across different projects or even organizations.

Acknowledgements

I am grateful to God for giving me breath and happiness. For his guiding throughout this long journey, providing me health, protection, and also wisdom to always seek inner peace and the peace of those who are part of my life.

I am immensely grateful to my father Wilson and mother Lucia, who raised me and taught to be a righteous person. For all the moments of advice and tenderness. I also thank my brothers João and Hugo and how I grew up to be a better man inspiring me through your examples as well as being an example to both of you. To Lisley, thank you for all the moments at my side, for being my companion through life and being kind and gentle in uncountable moments.

To Franklin and Wilkerson, two extraordinary professors, who by lucky are my advisors during the master degree. I am greatly grateful to be your student and to have learned a small parcel of your knowledge. Thanks for all the guidance, advices, patience, and also misleading "*the*"s.

To all SPLab/e-Pol members for their friendship and help during the master degree. Specially, to Alysson, Adriana, Bruno, Dalton, Izabela, Katyusco, Luiz Augusto, Lilian, and Matheus Gaudencio. Thanks for all the advices, guidance and friendship, making my days lighter and pleasing.

I thank all close friends, who shared moments of joy and sadness and helped me in so many uncountable ways. Special acknowledgments to Alcione Pinheiro, Andryw Marques, Arthur Felipe, Barbara Tsuyuguchi, Diego Tavares and Elisa, Heloá Aires and her family, Nathália Italiano, Ronaldo Regis, Simone Alciole and Carlinhos, Thiago Batista and his family, and also Von Brauner and his family. I also thank my RPG friends, Fabiano, Felipe Leal, Ítalo, Uian, and Wendel as well as past Accenture friends Abraão Morais, Anderson Pablo, Arthur Navarro, Burns, Diego Maia, Francisco Barros, Francisco Chiquinho, Gustavo Rocha, Hamon Barros, Jailson Pereira, Lobinho, Lorena Lira, Rafael Vilaca, Renato Ferraz and so many others. Best regards.

In closing, I thank the Federal University of Campina Grande and the Federal Police of Brazil for their financial support (by the agreement Nº 754664/2010) as well as all COPIN staff members, which promptly helped me throughout the course.

To my grandfather, who until his last breath was a passionate and wise man.

Contents

List of Symbols

AHP - *Analytic Hierarchy Process*

CASE - *Computer-aided Software Enginneering*

CoEST - *Center of Excellence for Software Traceability*

COST - *Commercial off-the-shelf*

DSL - *Domain Specific Language*

OCL - *Object Constraint Language*

OO - *Object Orientation*

RE - *Requirements Engineering*

RT - *Requirements Traceability*

SOA - *Service Oriented Architecture*

SQL - *Structured Query Language*

TQL - *Trace Query Language*

TracQL - *Traceability Query Language*

TRL - *Traceability Representation Language*

UML - *Unified Modeling Language*

VTML - *Visual Trace Modeling Language*

XML - *Extensible Markup Language*

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Requirements traceability refers to the ability to describe and follow the life of a requirement, in both a forwards and backwards direction (*i.e.* from its origins, through its development and specification, to its subsequent deployment and use, and through all periods of on-going refinement and iteration in any of these phases) [21].

In order to achieve traceability, a traceable environment must be established. Such environment is composed of procedures, methods, techniques and tools to accomplish the traceability process [43]. Usually, this environment is built upon a **model** that gives a uniform view for traces and traceable objects. The traceability model [43] considers three main aspects, which encompass the definition, production, and extraction of trace links. The *definition* elucidates what type of artifacts should be traced and how traces are represented; the *production* produces traces according to defined strategies and techniques; and finally, the *extraction* provides different and flexible ways to retrieve the produced traces.

1.1 Benefits of Requirements Traceability

Once requirements traceability is established, an organization may benefit from its support in different areas [2, 17, 31, 45, 55, 63], *e.g.* project management, process visibility, verification and validation, and maintenance. For instance, requirements traceability:

1. Simplifies project estimates, making project management easier. By following traceability links, a project manager can see how many artifacts will be affected by a change request, thus she/he can measure its cost;

2. Improves process visibility to both project engineers and customers. In such context, engineers and customers have access to contextual information about the life-cycle of requirements, which assist them in determining the origins of a requirement, its importance, how it was implemented, and how it was tested;

3. Allows verifying whether a system complies with its requirements and that they have been implemented accordingly. Hence, supporting verification and validation activities;

4. Makes it easy to determine what requirements, design, code, test cases, and other requirement related artifacts need to be updated to fulfill a change request made during the software project's maintenance phases [2, 55].

Regarding aforementioned benefits, we emphasize the importance to trace requirements to **system artifacts**. As a myriad of artifacts are produced during the life-cycle of a project, by tracing requirements to such artifacts one may: *(i)* identify the correct parts of the system to generate or reference appropriate test data; and *(ii)* determine which parts of the system (artifacts) would be impacted due to changes in one or more requirements. As a consequence, collateral effects to the deliverable software can be minimized if all artifacts are correctly traced. Thus, better assuring the deliverable's quality [2, 17, 31, 45, 55, 63].

1.2 Requirements Traceability Challenges

Besides requirements traceability benefits have been acknowledged [55,58], there are factors that hinder its correct utilization [3,16,21,27,46]. For instance, there is not a consensus about the traceability process [27, 46] and, as a consequence, requirements traceability practices cannot be unified across different organizational settings [16]. Moreover, as the complexity of developed systems increases, the myriad of artifacts and traceable information also increases. Thus, one is encumbered of tracing requirements through different abstraction levels and through heterogeneous traceability processes, which vary according to organizational settings [21, 22, 55]. As a consequence, throughout many organizations, requirements traceability is a singular, burdensome, and time-consuming activity.

In order to address the aforementioned issues, the present work considers the grand challenges of requirements traceability, proposed by Gotel *et al* [22]. By classifying research contributions and tracking progress in the field, the challenges discuss the necessity *(i)* to promote requirements traceability based on abstractions, rather than concrete artifact types; and *(ii)* to identify and standardize key aspects of the traceability process.

The necessity to promote requirements traceability based on abstractions is related to **scalable traceability**, *i.e.* inhibiting limits to what type of artifacts can be traceable. Considering heterogeneous sources of artifacts that are likely to arise in a project, one is encumbered of the task to understand the particularities of each type of traced artifact through different abstraction levels [22]. Therefore, benefits provided by requirements traceability, such as identifying the correct parts of the system to be tested or determining which artifacts would be impacted due to changes are compromised due to the increasing overhead of understanding each type of traced artifact.

Moreover, the necessity to identify and standardize key aspects of the traceability process is related to **portable traceability**, *i.e.* how requirements traceability techniques can be used across different projects or even organizations. The lack of a consensus about the traceability process precludes that requirements traceability practices can be unified across different organizational settings. As a consequence, traceability practices, techniques or process cannot be ported. For instance, suppose that a hospital management system called EasyClinic [15] is being developed and traced by an organization `A`, which has a specific traceability process (Figure 1.1). However, due to cost limitations and contract clauses, after being developed, the maintenance of the system is carried over to organization `B`, whose traceability process and trace link representation differ from `A`. In such context, without a uniform representation of trace links and also without an overall traceability process, the organization `B` cannot exploit the already extracted trace links or even the process which produced them. Thus, the heterogeneity of traceability processes as well as trace link representations preclude portable traceability.

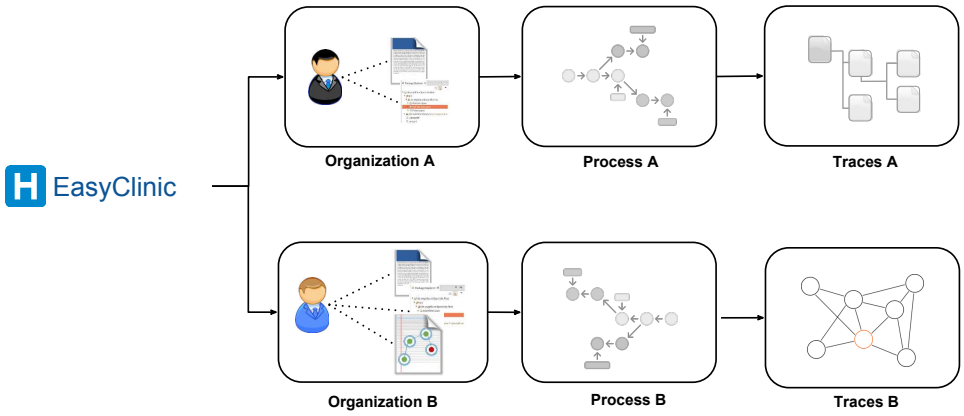


Figure 1.1: Hindered Interoperability of Traceability Processes

1.3 Objective

Considering the factors that hinder requirements traceability practices, the present work objectives are twofold *(i)* to provide an abstraction to traceable information; and *(ii)* to identify key aspects of the traceability process considering the traceability model and also tracing requirements to system artifacts. To this extent, we propose a Traceability Representation Langauge (TRL); and also propose and detail a traceability process that exploits the defined language.

Therefore, by means of the proposed representation (TRL) and process, the present work proposes an approach for abstracting requirements traceability and tracing requirements to system artifacts.

1.4 Scope

Considering defined objectives (Section 1.3), the present work restrains its contributions to two research fields: *(i)* requirements traceability languages and *(ii)* requirements engineering process improvements.

Regarding the definition of abstractions to requirements traceability, our work is related to requirements traceability languages [33,34,45,57]. Notice that, identifying which artifacts or elements should be traced as well as providing means of abstracting this information is not a trivial task. Several traceability languages provide abstractions in different levels, such

as requirements rationale [45], models [33, 45], or system artifacts [34, 57]. Even though, specifying the concept of granularity and providing a way to define and retrieve artifacts in these different levels of abstraction is challenging [22]. Therefore, TRL contribution is restrained to providing abstractions to requirements and system artifacts as well as providing means of retrieving them.

Additionally, by means of identifying key aspects of the requirements traceability process, this research is related to requirements engineering process improvements. Even though, process improvements in requirements engineering may consider different aspects such as improving how system requirements are elucidated [24, 40, 59], how to understand the dynamics and particularities of such process [20, 53, 61], how to improve requirements traceability [22, 51], and so forth. Thus, our proposed process is restrained to the scope of requirements traceability practices. Considering *(i)* the three major phases of the traceability model (i.e. definition, production, and extraction); and *(ii)* tracing requirements to system artifacts in forward and backward directions, we observe how traceability techniques, methods, and tools can be ported to different contexts. Thus, portable traceability can be addressed.

1.5 Envisioned of Contributions

As a consequence of the objectives defined in Section 1.3, the envisioned of contributions of the present work are detailed as follows:

- **A traceability representation language (TRL).** TRL is a declarative language which defines trace links as a relationship between requirements and artifacts. Through language's constructions, one can specify traced requirements and artifacts through a single abstraction. Moreover, TRL provides the construction of traceability queries, that support searching and retrieving requirement related artifacts;

- **A requirements traceability process.** TRL is exploited by a requirement traceability process, centered on the traceability model. The proposed process encompasses the three phases of the traceability model. In the definition phase, the process considers the language specification as an input and the set of artifacts to be traced is defined according to stakeholders' needs. Then, in the production phase, artifacts are analyzed

and tagged. Hence, trace links are produced (according to language's constructions) and grouped for later query and also maintenance. Finally, in the extraction phase, requirements are queried (through language's queries) and requirement related artifacts are retrieved; Moreover, in order to provide interfaces to the traceability activities and establish means of communication through the different activities/phases of the proposed process, requirement traceability contracts are defined;

- **A requirements traceability tool**, namely SORTT, which supports the proposed approach and automates the production and extraction of trace links by means of different traceability services. SORTT relies on the defined traceability contracts and encompass distinct phases of the proposed process;

- **Evaluation of the proposed approach.** In order to evaluate the proposed approach, we considered benchmarks from the Center of Excellence for Software Traceability (CoEST)[1] as well as an industrial project being developed at the Software Practices Laboratory for the Federal Police of Brazil. Considering such projects, the evaluation was twofold: *(i)* the proposed language was evaluated considering its readability and writability, *i.e.* how comprehensible it is; and *(ii)* the proposed process was evaluated regarding its performance and effectiveness, *i.e.* how well it supports requirements traceability tasks. As a result, we observed that the language's constructions were evaluated as easily read/written and also that its queries support requirements traceability tasks. Therefore, the proposed language is a feasible approach to provide an abstraction to requirements traceability. Moreover, we observed that, in comparison with an ad hoc process, the proposed process improves the performance and efficiency of requirements traceability, while maintaining the same accuracy of other approaches. Therefore, the proposed approach is feasible to address abstractions to requirements traceability as well as foster the discussion of major aspects of the requirements traceability process, thus portable and scalable traceability can be addressed.

[1] `http://www.coest.org/`

1.6 Dissertation Outline

The remainder of the dissertation is organized as follows:

- **Chapter 2: Background** provides the theoretical background related to the main concepts discussed in this work. Thus, concepts related to requirements, artifacts, and requirements traceability are presented;
- **Chapter 3: Traceability Representation Language** presents our approach to represent traceable information through a Traceability Representation Language (TRL);
- **Chapter 4: Traceability Process** presents and details a requirements traceability process that underlies on the whole traceability model and the proposed traceability representation language and contracts;
- **Chapter 5: Tool Support** presents the design and implementation of the tool support for our proposed approach (language and process);
- **Chapter 6: Evaluation** details how the proposed traceability representation language and the proposed traceability process were evaluated;
- **Chapter 7: Related Work** discusses the fundamental works which are related to ours, either as a basis for our work or for its comparison;
- **Chapter 8: Conclusions** presents our final remarks, limitations, and prospects for future work.

Chapter 2

Background

In this chapter, we provide the theoretical background related to the main concepts discussed in this dissertation. To this extent, we firstly introduce concepts related to requirements and artifacts. Thus, the adopted definition of requirements traceability is presented and then, major concepts related to it are also discussed.

2.1 Requirement

A **requirement** is a documented description of what a system should do – the services that it provides and the constraints on its operations. A requirement reflects the needs of customers or stakeholders for a system and contains a series of information necessary to its correct conception [52]. Regarding the aforementioned definition, it encompasses both functional and non-functional requirements.

Functional requirements are statements of services that the system should provide, how the system should react to particular inputs, and how it should behave in particular situations. On the other hand, **non-functional requirements** are constraints on the services or functions offered by the system, which are often applied to the system as a whole, rather than individual system features or services [52]. As an example, Table 2.1 presents both functional and non-functional requirements for a hospital management system called EasyClinic[1] [15]. Both booking visits and changing reservations are examples of functional requirements. On the

[1]Minor modifications were made to the system in order to fully present all the concepts discussed through this dissertation.

Table 2.1: EasyClinic Requirements

Requirement	**Description**
REQ014	The system shall book visits
REQ015	Operators may request changes on reservations
REQ091	System's database management shall be implemented using Oracle 11g

other hand, strictly specifying Oracle 11g as the system's database management is a non-functional requirement.

Regarding the requirements briefly presented in Table 2.1, it is reasonable that they should be further specified, thus one can establish the basis for agreement between customers and contractors for the software to be developed. Considering that there are many different software development methodologies [8, 30], there are also different ways to specify requirements. For instance, requirements can be detailed through user stories, using natural language sentences or a structured natural language, through mathematical or graphical notations, and so forth [44, 52]. Nevertheless, best practices, extracted from IEEE Std 830-1998 [26], state that a software requirements specification (SRS) should be:

1. **Correct** – if, and only if, every requirement stated therein is one that the software shall meet. Notice that, there is no tool or procedure that ensures correctness. Usually, the customer or stakeholders determine if the specification correctly reflect their needs. In such context, traceability makes this procedure easier and less prone to errors;

2. **Unambiguous** – if, and only if, every requirement stated therein has only one interpretation. As a minimum, this requires that each characteristic of the final product is described using a single unique term;

3. **Complete** – if all significant requirements, whether functional or non-functional, are included in the specification;

4. **Consistent** – if there is no conflict between any subset of individual requirements described within the specification;

5. **Ranked for importance and/or stability** – if each requirement has an identifier to indicate either the importance or stability of that particular requirement. For instance,

some requirements may be essential, especially for life-critical applications, while others may be desirable;

6. **Verifiable** – if, and only if, every requirement stated therein is verifiable. A requirement is verifiable if there exists some finite cost-effective process with which a person or machine can check that the software product meets the requirement;

7. **Modifiable** – if the specification is structured such that any changes to the requirements can be made easily, completely, and consistently while retaining the structure of the specification. A modifiable specification usually has a coherent and easy-to-use organization, is not redundant and express each requirement separately;

8. **Traceable** – if the origin of each requirement is clear and if the specification facilitates either the referencing of each requirement in future development or the enhancement of the documentation. In such context, traceability is especially important when the software product enters the operation and maintenance phase. As code and design documents are modified, it is essential to be able to ascertain the complete set of requirements that may be affected by those modifications.

In the scope of this work, we are specially interested on aspects related to the verifiability of the software product and also the traceability of its requirements. Since one of the primary measures of success of a software system is the degree to which it meets the purpose for which it was intended [40], traceability can be of fundamental importance in such context (as Section 2.3.1 details).

2.2 Artifact

Throughout this dissertation, an **artifact** is any element produced in a software development project. They are originated from heterogeneous sources, such as system requirements, use cases, source code files, test cases, and so forth. For instance, Table 2.2 presents an excerpt of artifacts related to the EasyClinic's requirement: booking visit. In such context, the requirements specification, which details the booking requirement, the source code files (classes) and also the test cases, which test this functional requirement, are all considered as artifacts.

Table 2.2: EasyClinic Artifacts

Requirement	REQ014: The system shall book visits
Classes	Booking BookingAgenda
Test Cases	TC01: Testing a booking in a visit day TC02: Testing a booking not in a visit day TC03: Testing a booking to a patient in an intensive care unit (ICU)

2.3 Requirements Traceability

Requirements traceability refers to the ability to describe and follow the life of a requirement, in both a forwards and backwards direction (*i.e.*, from its origins, through its development and specification, to its subsequent deployment and use, and through all periods of on-going refinement and iteration in any of these phases.) [21].

In the software development life-cycle, a lot of information is usually used and produced. They are originated from heterogeneous sources, such as system requirements, source code, test cases, and so forth. Nevertheless, one of the primary measures of success of a software system is the degree to which it meets the purpose for which it was intended [40]. Therefore, verifying if a system meets its requirements is one essential task in the software development process. Such task is supported by requirements traceability. It provide means to trace requirements from its origins, through its development and specification, to its subsequent deployment and use, and through all periods of ongoing refinement. Therefore, requirements traceability is an essential task to the software development process.

2.3.1 Importance of Traceability

Typically, as the system evolves, its requirements change over time. New functional requirements are added, updated or deleted and also new constraints, or non-functional requirements, are specified. For instance, the traffic of a web social system could increase due to new incoming users. In such context, one may require that the login and search operations may take no longer than 3 seconds. Moreover, as the quantity of users increases, one may require new search filters, thus users may find others more efficiently. In such context, the

cost of changing a requirement increases dramatically over the life-cycle of a system [9, 32].

Since maintenance usually encompasses changes in the requirements, it is critical for stakeholders to understand requirements before making any changes to the system, and also using the knowledge about requirements on making critical decisions about the system, design decisions and it maintenance [63]. In this context, requirements traceability may support:

- Change impact analysis: determining which parts of the system would be impacted due to changes in one or more requirements [2, 55];
- Program Comprehension: understanding the relations between requirements as well as the capture, tracking and evolution of requirements, in order to comprehend the overall system evolution [42, 55];
- Consistency checking: determining if changes to the system have created unnoticed and unintended contradictions to the traced requirements [17, 31, 45, 55];
- System testing: understanding requirements to identify the correct parts of the system to generate or reference appropriate test data, and to check if tests properly cover all requirements as well as checking standards compliance [2, 21, 55].

2.3.2 Trace Link

According to Gotel and Finkelstein a **trace**, or a **trace link**, is a relationship between a requirement and an artifact [21]. It is noteworthy that based on this definition, an artifact can be related to one or more requirements according to different types of relationships. In such context, an **element** represents the different parts, entities, and objects present in this relationship. Thus, elements are requirements as well as artifacts, such as use cases, source code files, test cases, etc [55].

Regarding trace links, it is important to emphasize that stakeholders with different perspectives, goals and interests may be interested in different **types of relations**. Therefore, existing approaches and tools for traceability support the representation of different types of relations between requirements and artifacts. In such context, Spanoudakis and Zisman

organize the various types of traceability relationships into eight main groups [55], which are further detailed as follows:

- **Dependency** relations, which state that an element `e1` depends on an element `e2`, if the existence of `e1` relies on the existence of `e2`, or if changes in `e2` have to be reflected in `e1`;
- **Refinement** relations are used to identify complex elements and how it is detailed, or further refined, by other elements;
- **Evolution** relations type signify the evolution of elements of software artifacts. In this case, an element `e1` evolves to an element `e2`, if `e1` has been replaced by `e2` during the development, maintenance, or evolution of the system;
- **Satisfiability** relations type, in which an element `e1` satisfies an element `e2`, if `e1` meets the expectation, needs, and desires of `e2`; or if `e1` complies with a condition represented by `e2`. Such relation type is usually used to establish constraints and pre-conditions between requirements;
- **Overlap** relations state that an element `e1` overlaps with an element `e2`, if `e1` and `e2` refer to common features of a system or its domain;
- **Rationalisation** relations, which are used to represent and maintain the rationale behind the creation and evolution of elements, and decisions about the system at different levels of detail;
- **Contribution** relations are used to represent associations between requirement artifacts and stakeholders that have contributed to the generation of the requirements;
- **Conflict** relation signifies conflicts between two elements `e1` and `e2`. Conflict relations are usually used to signify conflicts between requirements, design decisions or components.

2.3.3 Traceability Modes

Based on the trace link definition presented in Section 2.3.2, the concept of requirements tracing is quite simple: to follow relationships or links [43]. Notwithstanding, there are sev-

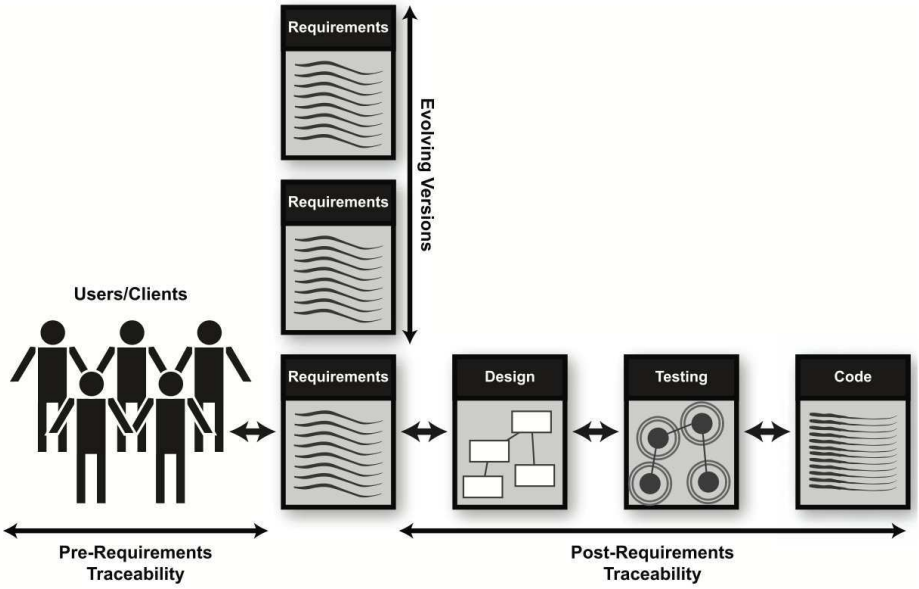


Figure 2.1: Software Requirements Traceability Overview

eral modes of traceability, which can assist different needs, such as the necessity to identify the test cases related to a given requirement or the necessity to identify conflicting requirements and their stakeholders. Therefore, we further detail traceability modes.

Figure 2.1, extracted from [27], summarizes the several ways in which requirements tracing can be performed. As regards the direction of tracing, a requirement may be traced in a **forward** or **backward** direction; as regards requirements evolution, a requirement may be traced to aspects occurring **pre** or **post** its inclusion in the requirements specification; and as regards the type of the objects involved, we may have **inter** or **extra-requirements** traceability. Forward traceability is the ability to trace a requirement to its subsequent generated artifacts. On the other hand, backward traceability is the ability to trace an artifact to its origin requirement. Furthermore, inter-requirements traceability refers to the relationships between requirements, whereas extra-requirements traceability refers to the relationships between requirements and other artifacts.

In the present work, whenever not specified, when we refer to requirements traceability (or tracing), its scope is limited to forward and backward traceability, usually in the context of extra-requirements traceability, *i.e.* we are interested on tracing requirements to their related artifacts or artifacts to their origin requirements.

2.3.4 Traceability Model

In order to achieve traceability, a traceable environment must be established. The **traceability model** is a central component of such environment, around which the tracing procedures, methods, and tools are organized [43], therefore requirements traceability can be accomplished. The traceability model encompasses three main phases, summarized in Figure 2.2. The **definition** elucidates what type of artifacts should be traced and how traces are represented; the **production** produces traces according to defined strategies and techniques; and finally, the **extraction** provides different and flexible ways to retrieve the produced traces.

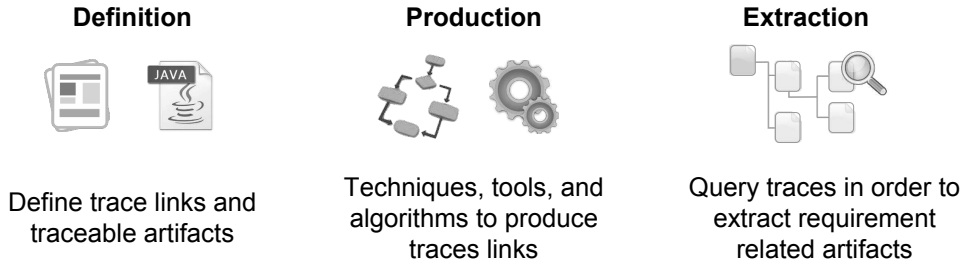


Figure 2.2: Traceability Model Overview

The traceability model should define its trace units, *i.e.* how the traceable objects are represented in the model [43]. To this extend, traces and which information they represent should be clearly defined. Notice that, traces are a core element in the whole traceability process. Once a trace link representation is defined, it will be created and manipulated through the traceability process in order to trace requirements and their diverse relationships.

The production encompasses the perception, registration and maintenance of trace links. Traces are produced according to the defined trace link representation. The generation of trace links can be **manual**, **semi-automatic** or **automatic** [55], and this process of traces' production is closely related to available techniques present in the literature [1, 4–6, 13, 14, 25]. For instance, traces can be manually produced by marking artifacts with identifiers to their related requirements or traces can be automatically produced using information retrieval techniques [1, 25], calculating term frequencies between requirement specifications and artifacts.

According to Spanoudakis and Zisman, the manual generation of trace links is normally supported by visualization and display tool components, in which the artifacts to be traced

are displayed and the users can identify and mark the elements which are related [55]. Examples of this approach occur in mostly industrial requirement management tools, such as IBM DOORS[2] or Jazz[3]. Despite the fact that manual approaches help on identifying the relationships between the traced artifacts, the effort to establish the relationships is still high, specially when dealing with large and complex systems [55]. In such scenario, the correctness of the traceability relations relies on *(i)* the understanding of the the system (and its artifacts) to be traced; and *(ii)* the user who identify them.

Semi-automatic approaches try to overcome the burden of manually identifying trace links. In this scenario, trace links are generated in a semi-automatic way. Users and automated processes interact in order to produce the trace links. For instance, users may register traced artifacts in an event server, thus whenever an artifact is updated all related artifacts are notified. Such approach is called event-based traceability [13, 14]. As a second example, one may conceive a rule-based engine to identify the relationships between artifacts, thus their relationships are automatically extracted [54, 56]. The aforementioned approaches provide improvements when compared with a manual approach, however the initial effort of establishing rules or registering artifacts in the event server may still cause the production of trace links to be error prone and time consuming [55].

In order to minimize the time and effort to produce trace links, one may produce them automatically. In this approach, automated processes delve into the traced artifacts and infer their relationships based on some comparison criteria. For instance, Antoniol et al. propose the usage of information retrieval (IR) techniques in order to automatically identify the relationships between requirements and source code files [1, 25]. Considering automatic approaches, they in fact minimize the effort to extract trace links. Even though, they introduce a new challenge, which is related to the trustworthiness of the automatic approaches. Therefore, one may inquire if the automatically produced trace links are correct.

Regarding the production of trace links, it is important to emphasize that none of the described approaches overcomes the other. Factors influencing the time and effort to establish trace links as well as organizational environments, or standards, may dictate the usage of a specific approach. Therefore, even with significant advancements in the field, requirements

[2] `http://www-03.ibm.com/software/products/en/ratidoor`

[3] `https://jazz.net/`

traceability remains a challenge [21, 22, 27].

Finally, the traceability model should define how traces are extracted. The extraction of trace links should consider the necessity which drove the tracing and thus, provide different and flexible ways to retrieve the traceable information [43]. To this extent, traces can be extracted **selectively**, identifying artifacts which matches certain selected patterns of objects and relations, or **interactively**, delving in a step-wise manner into traces and their relations and inquiring the ones that are most likely related to the task at hand.

Chapter 3

Traceability Representation Language

In this chapter, we present an approach to represent traceable information through a Traceability Representation Language (TRL) [36, 39]. TRL is a declarative language. Requirements, types of relationships, artifacts and their types are declared through language's constructions. In turn, their relationships are retrieved through TRL's specified queries.

In order to detail the proposed TRL and since trace links are a central artifact in the whole traceability process, we firstly discuss TRL's trace link representation (Section 3.1). Then, we present language's constructions (Section 3.2) and queries (Section 3.3). Finally, a complete example is presented (Section 3.4) and then, we present the chapter debriefing (Section 3.5).

3.1 Trace Link Representation

To define a trace link, we consider the most adopted traceability definition in the literature, proposed by Gotel and Finkelstein [21], which states that a trace relates requirements and artifacts. Thus, TRL defines trace links as relationships between **artifacts**, of some **type**, and **requirements**. As each artifact can be related to a requirement in different ways, such links also have a **type of relationship**.

Considering the aforementioned definition, Figure 3.1 presents TRL's abstract syntax [1].

[1] The usage of a UML meta-model to represent the grammar's abstract syntax is a recurring standard in the literature and it favors a holistic view of language's elements and their relationships. Therefore, its adoption [49].

Notice that a trace link associates a requirement and an artifact. Also, it has a specific relation type, which is derived from some of the relation types discussed through the literature [55], and presented in the Chapter 2. Notice that, such enumeration is not complete, and it can be further increased according to the different types of relationships that are likely to emerge in the context of requirements traceability.

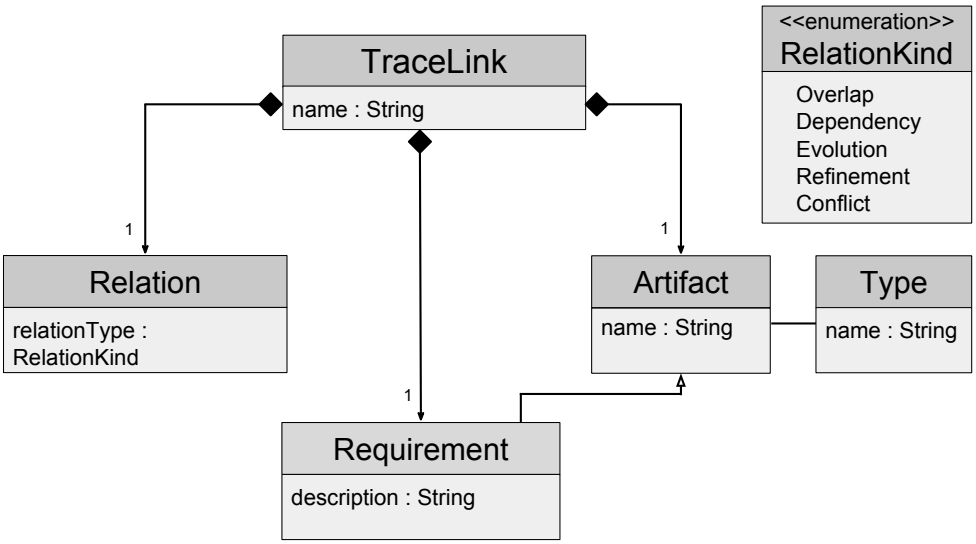


Figure 3.1: Trace Link Representation Abstract Syntax

As a concrete syntax example, let us consider the EasyClinic system, presented in Chapter 2. Listing 3.1 presents a trace link between one system requirement and one source code file. The trace link `t1` relates the booking requirement (`REQ014`) to the `Booking` class, with an `Overlap` relation type, *i.e.* the class implements features described by the requirement [55]. On the other hand, the trace link `t2` relates the booking requirement (`REQ014`) to the `TC01` test case, with an `Dependency` relation type, *i.e.* the test case shall be revised whenever there are changes in the requirement [55]. Finnally, the trace link `t3` state that there is relation of conflic between requirements `REQ015` and `REQ014`, *i.e.* if both requirements are taken into account, the system's state may become inconsistent [55].

Listing 3.1: TRL - Trace Link Representation

```
1 tracelink t1 = {REQ014, Overlap, Class, Booking};
2 tracelink t2 = {REQ014, Dependency, TestCase, TC01};
3 tracelink t3 = {REQ015, Conflict, Requirement, REQ014};
```

3.2 Language's Constructions

TRL's constructions compromise the major elements present in a trace link, *i.e.* requirements, artifacts, artifact types and also types of relationships. Such elements are summarized in Figure 3.1 and detailed in this section.

Requirements, either functional or non-functional, are declared using the `requirement` keyword. A requirement has a name, or an identifier, which facilitates its identification and relates it to its requirement specification. Such construction can have an optional description field, further detailing the declared requirement. For instance, Listing 3.2 presents the booking requirement according to TRL's constructions (line 1). First, the requirement is declared according to its ID (`REQ014`), then its description field further details that this ID is in fact the `booking visit` requirement.

Artifacts are declared through the `artifact` keyword. They are constructed with two parameters: *(i)* the first one, is the type of the artifact, which needs to be previously declared through the `type` construction, and *(ii)* the second one is the name or the identifier of the artifact itself. As an example, Listing 3.2 presents two artifacts and their types (lines 2-5). First, *(i)* a `Class` type is declared representing all traced artifacts related to source code files (line 2); and *(ii)* a `TestCase` type is also declared representing all traced artifacts related to test cases (line 3); Then, the artifacts themselves are declared. The `Booking` class is declared in line 4 and the test case `TC01` is declared in line 5.

Additionally, the possible trace link relationships are declared using the `relationtype` keyword (lines 6-7). If types of relationships are explicitly declared, one can comprehend all project's existing types of trace links even before querying them. In such context, Listing 3.2 presents a `Dependency` relation type (line 6) as well as a `Overlap` relation type.

Listing 3.2: TRL - Language's Constructions

```
1 requirement REQ014 = {"Book Visit"};
2 type Class;
3 type TestCase;
4 artifact (Class, Booking);
5 artifact (TestCase, TC01);
6 relationtype Dependency;
7 relationtype Overlap;
```

3.3 Queries Specification

In addition to the declaration of requirements, types, relation types, and artifacts, TRL also supports the specification of queries, that are used in order to retrieve and filter trace links. The overall query abstract syntax is presented in Figure 3.2. A query has a name, a body (formed by a query expression with operators and operands) and, optionally a series of parameters. Considering passed parameters as filters, the body defines how the result set of the query will be retrieved.

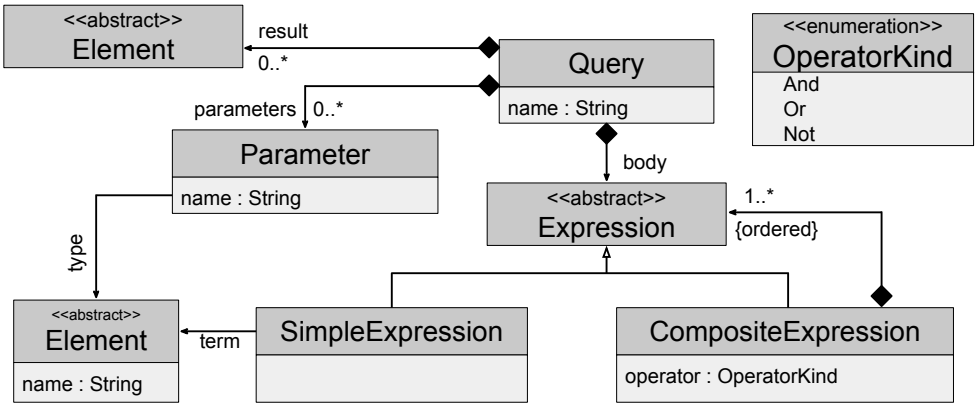


Figure 3.2: Query Abstract Syntax

For instance, consider the task of retrieving trace links related to the booking requirement. In Listing 3.3, the query `findRelated` receives a requirement parameter `r` and states that the set of trace links to be retrieved, or the `result` of the query, is filtered by this `r` parameter.

Listing 3.3: TRL - Simple Query

```
1 query findRelated (requirement r) { result r; }
```

Also, expression bodies support the declaration of composite expressions, which are formed using logical operators `and`/`or`/`not`. For instance, the query `findRelated` could be overloaded in order to *(i)* retrieve all trace links with a specific artifact type related to a given requirement (line 1); or *(ii)* query all trace links with a specific type of relationship, which are related to a given requirement (line 3).

Listing 3.4: TRL - Composite Queries

```
1 query findRelated (requirement r, type t) { result r and t; }
2
3 query findRelated (requirement r, relationtype s) { result r and s; }
```

As the complexity of one requirement traceability task increases, more elaborated queries can be constructed. For instance, the query `findMultipleRelatedTraces`, shown in Listing 3.4, retrieves trace links related to two given requirements (`r1` and `r2`), which are filtered by a given artifact type (`t`). Nevertheless, querying trace links through different parameters such as requirements, artifacts, artifact types, and types of relationships provide a feasible mechanism to the extraction phase, present in the traceability model.

Listing 3.5: TRL - Complex Queries

```
1 query findMultipleRelatedTraces (requirement r1, requirement r2, type t)
2 { result r1 and r2 and t; }
```

3.4 Example

As a complete example, let us consider the EasyClinic system, presented in Chapter 2 and also throughout Sections 3.1, 3.2, and 3.3.

Listing 3.6 presents TRL constructions of this project, which could be manually declared, or extracted by means of a traceability process (Section 4). Based on language's declarations, it is possible to identify system's functional requirements (lines 1-2), such as booking visits (`REQ014`) and changing reservations (`REQ015`), or non-functional requirements (line 3) as the constraint on the system database (`REQ091`). Considering language's declarations, it is also possible to identify traced artifact types (lines 5-6), or even artifacts (lines 8-12). Therefore, through TRL's constructions, a requirements engineer can identify all groups of traced artifacts as well as single artifacts, and their related types, in a unified manner. For instance, one could identify that classes (line 5) and test cases (line 6) are traceable artifacts, and that during project's implementation, they are concretized through the `Booking` and `BookingAgenda` classes as well as several test cases, such as `TC01` to `TC03`. Additionally, considering trace link relation types (lines 14-15), it is possible to reason that project's artifacts are related through `Dependency` and `Overlap` relationships.

As a final remark, Listing 3.6 also presents the specification of a traceability query (line 17). Considering the necessity to trace requirements to artifacts of a specific type, the query `findRelated` provides means by which trace links can be searched and retrieved. Therefore, one could exploit the declared query in order to automatically search for trace related artifacts.

Listing 3.6: TRL - EasyClinic Example

```
1  requirement REQ014 = {"Book Visit"};
2  requirement REQ015 = {"Change reservations"};
3  requirement REQ091 = {"Oracle 11g database"};
4  ...
5  type Class;
6  type TestCase;
7  ...
8  artifact (Class, Booking);
9  artifact (Class, BookingAgenda);
10 artifact (TestCase, TC01-Testing a booking in a visit day);
11 artifact (TestCase, TC02-Testing a booking not in a visit day);
12 artifact (TestCase, TC03-Testing a booking to a patient at ICU);
13 ...
14 relationtype Dependency;
15 relationtype Overlap;
16
17 query findRelated (requirement r, type t) { result r and t; }
```

Regarding the EasyClinic example, presented in Listing 3.6, we emphasize language's syntax. The TRL's syntax, written according to the Backus-Naur's notation [23] is presented in Listing 3.7. Roughly, TRL is composed of five major expressions, which define requirements (lines 4-5), types (line 7), artifacts (lines 9-10), relation types (line 12), and also queries (lines 14-24).

3.5 Chapter Debriefings

Requirements traceability refers to the ability to describe and follow the life of a requirement, in both forwards and backwards direction [21]. In order to achieve traceability, a traceable environment must be established. Such environment is composed of procedures, methods, techniques and tools to accomplish the traceability process. Nevertheless, requirements traceability usually involves delving into a myriad of trace links and artifacts. Since

artifacts can have different natures, such as requirements specification, source code files, test cases, and so forth, one is encumbered of tracing a requirement through different abstractions. Thus, it can be a burdensome, time consuming and elusive task [3, 21, 27, 46].

In order to provide a unified representation for expressing traceability information, requirements traceability grand challenges [22] discuss the necessity of *(i)* traceability based on abstractions, rather than concrete artifact types; and *(ii)* search, retrieval and filtering capabilities in order to assist traceability tasks. Therefore, the proposed traceability representation language can provide abstractions to artifacts, requirements and trace links as well as provide declarative queries through which requirements traceability tasks can be accomplished. Hence, the proposed language is a feasible approach to provide an abstraction to requirements traceability.

Listing 3.7: TRL Backus-Naur's Notation

```
1  <tlrLanguage> ::= <tlrElement>*
2  <tlrElement> ::= <requirementExpr> | <typeExpr> | <artifactExpr> | <relationExpr> | <queryExpr>
3
4  <requirementExpr> ::= "requirement" IDENTIFIER [<requirementDescription>] ";"
5  <requirementDescription> ::= "=" "{" STRING "}"
6
7  <typeExpr> ::= "type" IDENTIFIER ";"
8
9  <artifactExpr> ::= "artifact" "(" <artifactDeclaration> ")" ";"
10 <artifactDeclaration> ::= IDENTIFIER "," STRING
11
12 <relationExpr> ::= "relationtype" IDENTIFIER ";"
13
14 <queryExpr> ::= "query" IDENTIFIER "(" [<queryParameters>] ")" "{" "result" <queryExpression> "}"
15
16 <queryParameters> ::= <simpleParameter> [<multipleParameters>*]
17 <multipleParameters> ::= "," <simpleParameter>
18 <simpleParameter> ::= <parameterType> IDENTIFIER
19 <parameterType> ::= "requirement" | "type" | "relationtype" | "artifact"
20
21 <queryExpression> ::= <simpleQueryExpression> | <compositeQueryExpression>
22 <simpleQueryExpression> ::= ["("] ["NOT"] IDENTIFIER [")"]
23 <compositeQueryExpression> ::= ["("] <queryExpression> <queryOperator> <queryExpression> [")"]
24 <queryOperator> ::= "AND" | "OR"
```

Chapter 4

Traceability Process

In this chapter, we propose and detail a requirements traceability process [37, 38] that underlies on the whole traceability model (*i.e.* definition, production and extraction). A traceability process should define means by which pluggable activities, techniques, tools, and methods interoperate with each other in order to achieve requirements traceability [16,43]. Therefore, such process should be structured based on the traceability model and also it should define contracts that each process' phase should complies with [22]. Hence, we thoroughly discuss process phases, activities, actors, responsibilities, and input/output artifacts.

In order to detail the proposed process, we first introduce its workflow (Section 4.1), detailing each process' phases, *i.e.* definition (Section 4.1.1), production (Section 4.1.2) and extraction (Section 4.1.3), and subsequently, we detail how process' phases interoperate according to defined contracts (Section 4.2). Finally, concluding remarks are discussed (Section 4.3).

4.1 Process Workflow

The proposed process is composed of three phases that are structured on the traceability model, presented in Chapter 2 (Figure 2.2). The overall process workflow is presented in Figure 4.1, in which activities are represented as round-cornered rectangles (parenthesis above the activities' name represent its actors), input/output objects are represented as rectangles, decisions are represented as diamond shapes, and finally swimlanes divide the process according to the traceability model phases. Mostly analogous to the traceability model, in the

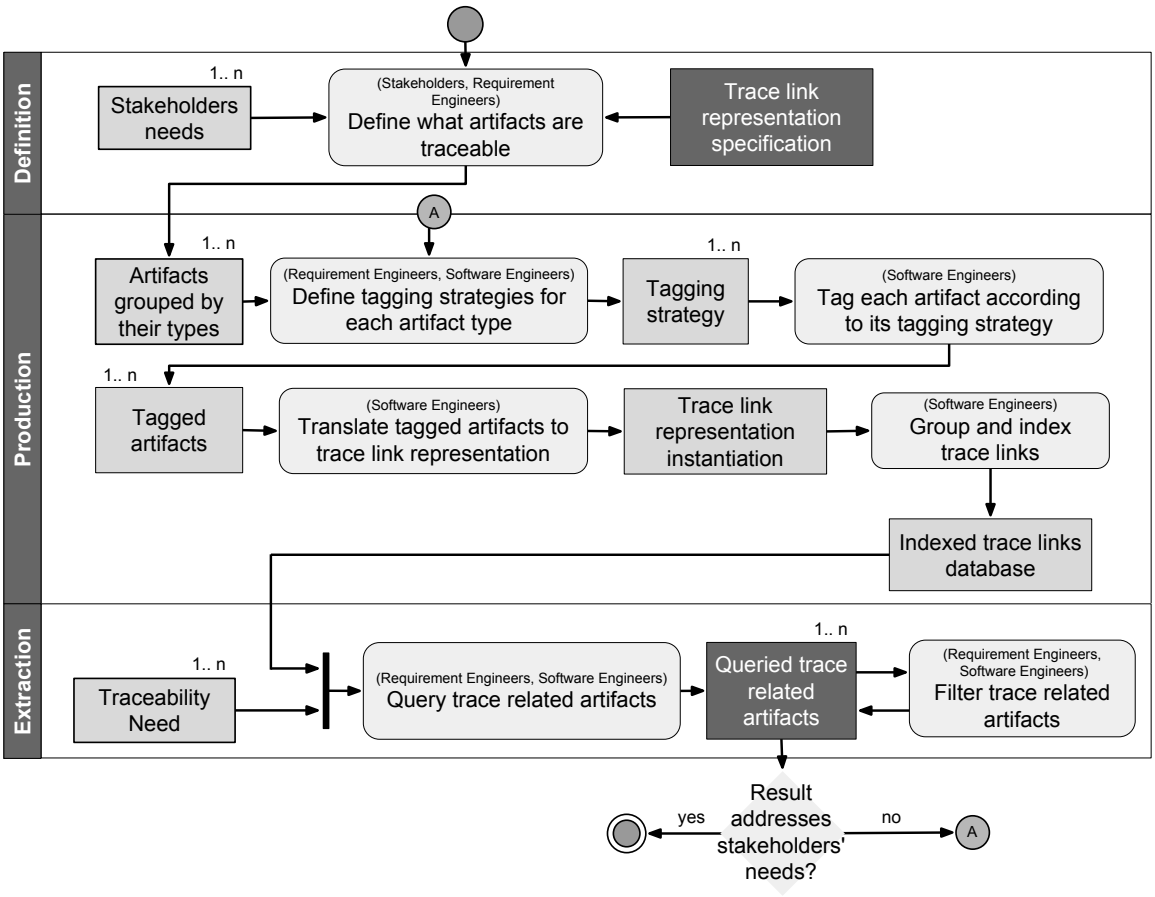


Figure 4.1: Traceability Process

definition phase, the set of traceable artifacts is defined according to stakeholders' needs. In the production phase, artifacts are analyzed and tagged. Then, trace links are produced and grouped for later query and also maintenance. Finally, in the extraction phase, requirements are queried and requirement related artifacts are retrieved. If extracted trace links are suitable, then the process is finished. Otherwise, a new iteration should be considered.

Regarding the proposed process and its phases, it is important to emphasize that its activities and their cost are amortized throughout the software development life-cycle [52]. Hence, its cost and effort are minimized through activities that are commonly carried over the conception of a system. As presentend in Figure 4.2, there are five major phases that encompass such life-cycle. First, requirements are elicited and, once specified, the system is designed and developed. Thereafter, it is tested in order to assure that it meets its requirements and then, system's maintenance and evolution should be taken into account. Considering the aforementioned phases, the traceability process definition phase is closely related to the re-

quirements specification phase. As requirements are specified, one should reason how to trace them. The production phase is related both the design and the software development phases. Through the system design and development, its artifacts are produced and related to their requirements. Finally, the extraction phase is related to the test and evolution phases. Thus, during software testing one can reason which tests are related to a specific requirement or in the evolution and maintenance phase, understand requirements evolution, conflicts or dependencies.

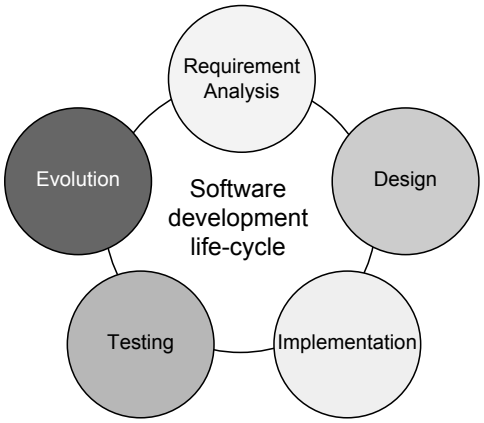


Figure 4.2: Software Development Life-Cycle

As a running example, throughout the next sections and subsections, we will discuss the proposed process, presenting how it is applied in the EasyClinic application context. In such scenario, consider that the system and, more specifically, its requirements and test cases are being traced due to the necessity of verifying if the system meets the purpose for which it was intended.

4.1.1 Definition

The definition phase is summarized in Figure 4.3. TRL's specification (Chapter 3) and stakeholders needs are considered as inputs to the activity of defining what artifacts are traceable, which produces the set of artifact types that need to be traced.

In the step ①, the activity of defining which artifacts should be traced is heavily supported by stakeholders feedback. Stakeholders responsibility is to elucidate functional and non-functional requirements, which artifacts will be produced as well as which ones should

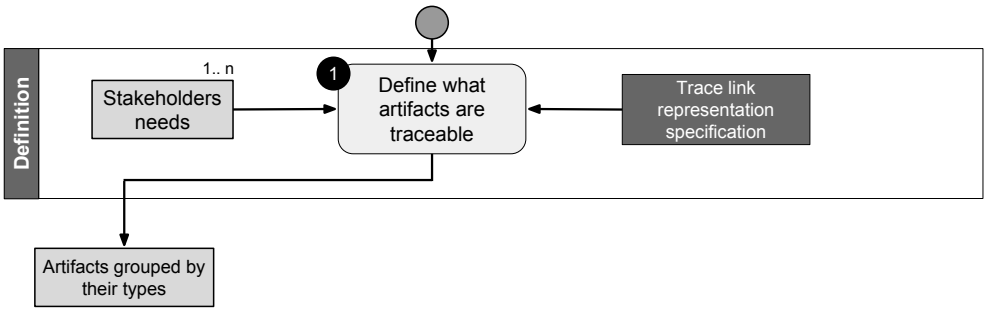


Figure 4.3: Traceability Process - Definition Phase

be traced. On the other hand, requirement engineers mediate the requirements specification process, clarifying divergent or conflicting requirements and also realizing how traceable artifacts comply with the adopted TRL.

Considering the TRL presented in Chapter 3, the trace link representation is one major element through the traceability process. Its data structure, presented in Section 3.1, will describe how trace links are represented. Thus, requirement engineers should reason about how traceable artifacts, and their relationships, will be translated to the adopted representation.

For instance, in order to evaluate if the EasyClinic system meets the purpose for which it was intended, stakeholder's needs state that requirements should be traced to test cases. Requirements are specified as plain text documents, as Listing 4.1 presents. On the other hand, test cases are written and structured using TestLink[1], an open source test management system, as detailed in Figure 4.4.

Listing 4.1: EasyClinic - Requirement Example

```
1 ID: REQ002
2 Description: Register Patient
3 Content: It allows the operator to meet request for a subscribing service, which will
    register a new patient in the system data base. A patient to be registered needs to
    provide some basic information such as his first and last name, address, contact number
    and national registration number. Once this information is provided, the subscribing
    service will validate them and register the patient, if the given information is valid.
    Otherwise, an error code and message will be returned be the subscribing service,
    detailing why the patient was not successfully registered ...
```

Regarding the aforementioned artifact types, the output of the definition phase is the project's set of traceable artifacts, grouped by their types. Notice that, some of these artifacts

[1] http://testlink.org/

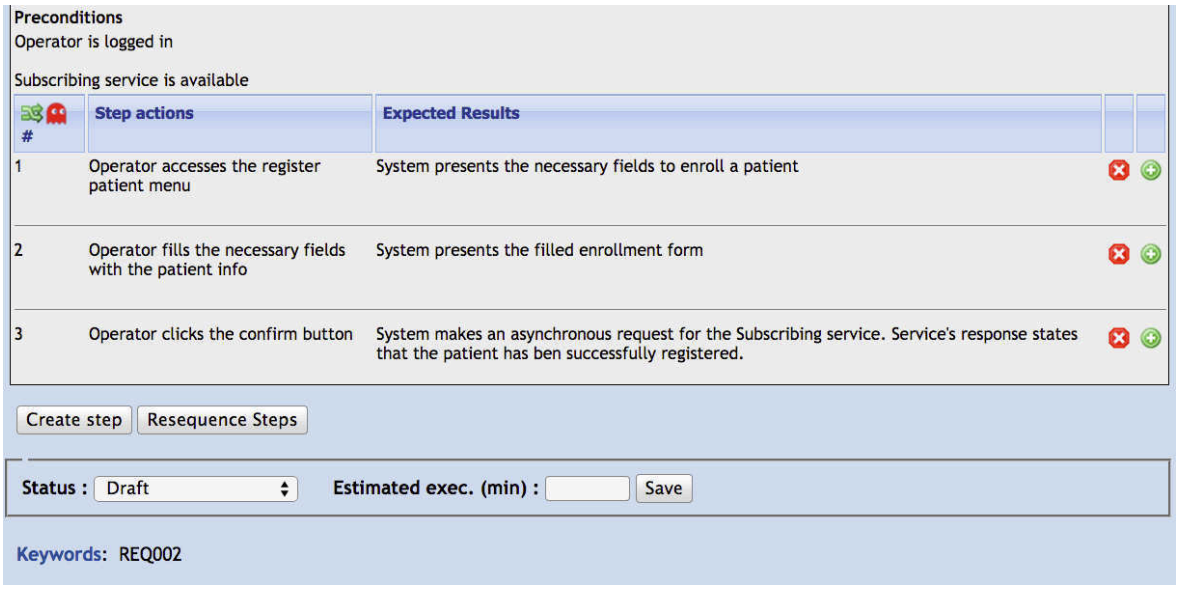


Figure 4.4: TestLink - TC01:Register patient with valid national ID number

are yet to be produced. Nevertheless, during artifacts' design and conception, activities of the production phase must be considered.

4.1.2 Production

Once the set of traceable artifacts is defined in the definition phase, it is necessary to plan how trace links will be produced. The production phase underlies on defining tagging strategies for each artifact type as well as defining how trace links will be produced, grouped and indexed for later query. Such phase is summarized in Figure 4.5.

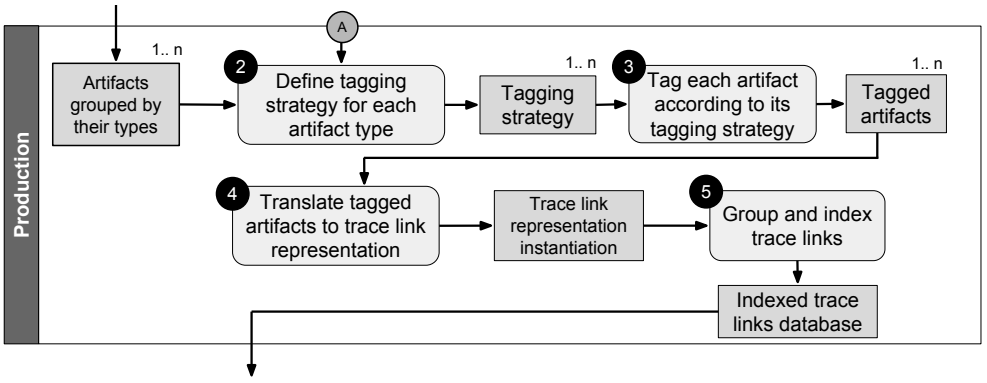


Figure 4.5: Traceability Process - Production Phase

First, in the step ②, a tagging strategy should be defined for each artifact type. The

strategy describes how to relate a requirement to a certain type of artifact. A tagging strategy is usually a plain text document, written in natural language, but structured such that its objectives are clear and there are no ambiguities. Regarding tagging strategies, it is reasonable to define strategies for each type of artifact, since different types of artifacts have different particularities, however some artifacts can share strategies. Tagging strategies can be *implicit* or *explicit*, and traceable artifacts' content should conform to their strategies. An explicit strategy defines a visible identifier that relates a requirement to the traceable artifact, whereas an implicit strategy establishes an intrinsic relationship between an artifact and a requirement. For instance, using a common vocabulary across different artifacts could be considered an implicit strategy, shared by different artifact types. On the other hand, a more explicit strategy would prescribe keywords to each artifact type. As an example, Listing 4.2 describes the adopted strategy to tag requirements and test cases in the EasyClinic system.

Listing 4.2: EasyClinic - Tagging Strategy Example

```
1 — In each requirement document, add the prefix REQ followed by three digits representing
    its sequential identifier, e.g. the second requirement "register patient" would be
    tagged as "REQ002 – Register Patient.doc";
2 — In each test case, add a keyword with the identifier of the requirement which is
    exercised by the test case. For instance, both test cases "Register patient with valid
    national ID number" and "Register patient with invalid national ID number" would have
    the keyword REQ002, of the register patient requirement.
3 — ...
```

Once tagging strategies are defined for each artifact type, in the step ③, artifacts are tagged according to their strategies. This is a straightforward process, although organizational policies and also internal reviews are encouraged in order to verify if artifacts conform to their strategies. For instance, in order to tag test cases, TestLink provides a keyword functionality, as presented in Figure 4.4. Thus, a keyword with the requirement identifier is added to each test case, which has a dependency to the referenced requirement.

Succeeding tagged artifacts, in the step ④, parsers should be provided for each strategy. They are responsible for translating trace links from tagged artifacts to the proposed representation (TRL). Contrasting the TRL, which is the input of the definition phase, the output of this activity is the instantiation of the trace links, according to the adopted representation. For instance, once the parser of the test cases specified on TestLink is run, the set of trace links relating test cases with their respective requirements is produced, such as presented in

Listing 4.3.

Listing 4.3: Easy Clinic - Produced Trace Links Example

```
1 tracelink t_1 = {REQ002, Dependency, Test Case, "TC01:Register patient with valid national ID
      number"}
2 tracelink t_2 = {REQ002, Dependency, Test Case, "TC02:Register patient with invalid national
     ID number"}
3 ...
4 t_n = { ... }
```

Finally, in the step ⑤, trace links are grouped and indexed according to a grouping and indexing strategy. Hence, trace links can be queried according to different factors. As part of this activity it is necessary to define a grouping and indexing strategy. Once a strategy is defined, trace links are grouped and indexing according to it. Grouping and indexing trace links facilitate overall trace comprehension and further analysis. For instance, Listing 4.4 presents a tree-like grouping data structure, in which traces are stored as forest of trees and each requirement is the root of a tree.

Listing 4.4: EasyClinic - Indexed and Grouped Trace Links Example

```
1  - REQ_002
2  -- Dependency
3  --- Test Case
4  ---- "TC01: Register patient with valid national ID number"
5  ---- "TC02: Register patient with invalid national ID number"
6  ...
7  - REQ_n
8  -- Type of Relationship
9  --- Type of Artifact
10 ---- Artifacts
```

Regarding grouped and indexed trace links, presented in step ⑤, it is important to emphasize that the proposed process allows both the update of existing trace links or their complete re-production, *i.e.* by re-executing the production phase of the traceability process, trace links maintenance might be addressed. Trace links maintenance is accomplished by means of reasoning which trace links were added, deleted, or updated to the trace links database [35]. Hence, through this approach only a fraction of the grouped and indexed trace links is altered. Notwithstanding, in order to assure the trustworthiness and the up to date of the trace links database, trace links maintenance also implies on additional costs and burdens to the proposed process [27, 46, 55]. Contrary to the trace links maintenance ap-

proach, the trace link re-production approach considers that all the trace links are completely updated throughout process' execution. By completely re-producing trace links, one can assure that they reflect the current state of the project [62]. Nevertheless, in order to mitigate costs related to the time and effort of this approach, automatic scripts or tools are required. Otherwise, the approach is impracticable [27, 46, 55].

Software engineers are the main actors of the production phase. However, requirement engineers and system architects have major roles while defining tagging strategies. Requirement engineers, system architects and software engineers should agree upon how to relate requirements to traceable artifacts. Thus, software engineers can execute the remaining activities of the production phase, either manually or automatically. As a final remark, it is also necessary to decide if the trace links should be maintained or produced. Such decision is closely related to organizational policies, project's deadlines and costs. Therefore, project's stakeholders should agree upon which strategy should be considered.

4.1.3 Extraction

The extraction phase underlies on querying, rendering and filtering trace links. Such phase is summarized in Figure 4.6.

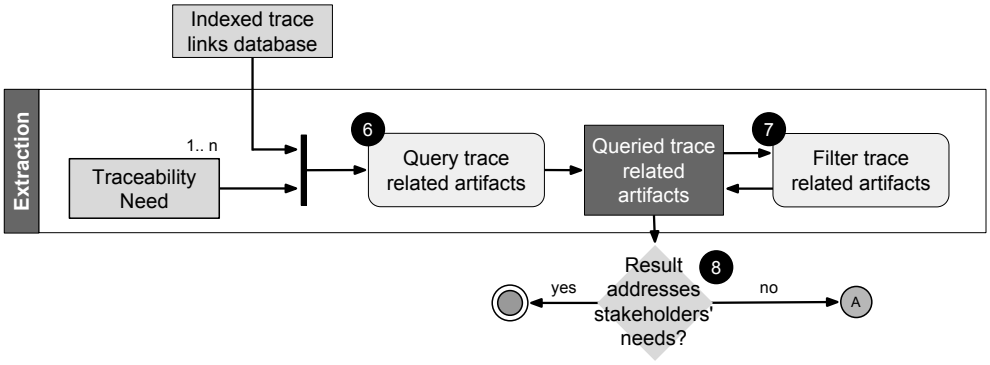


Figure 4.6: Traceability Process - Extraction Phase

The query activity, detailed in the step ⑥, should consider any element present in a trace link as queryable. Since tracing is an auxiliary activity to achieve some goal [43], providing a flexible search mechanism is essential. As traces are retrieved, it is necessary to present them. Therefore, as an output of the querying activity, the queried traces are rendered. Rendering

the query output provides mechanisms to facilitate the overall comprehension of the trace links and also their possible relationships. For instance, Listing 4.5 presents the set of extracted test cases related to the register patient requirement (REQ002), previously presented in Listing 4.3. It is important to highlight that, in this example, the result is rendered as plain text, using the trace link representation data structure. Notwithstanding, other data structures could be used to render the result such as requirement traceability matrices, graphs, and so forth [43, 55].

Listing 4.5: Easy Clinic - Extracted Trace Links Example

```
1 tracelink t1 = {REQ002, Dependency, Test Case, "TC01:Register patient with valid national ID
      number"}
2 tracelink t2 = {REQ002, Dependency, Test Case, "TC02:Register patient with invalid national
      ID number"}
```

In the step ⑦ of the extraction phase, one can filter the set of extracted trace links according to the elements present in the trace link data structure. Considering that a myriad of artifacts can be related to a requirement, it is reasonable that they can be filtered according to the task at hand. Therefore, the set of retrieved traces can be successively filtered until the desired traces are identified. As an example, suppose that one is encumbered of filtering all exception flows related to the register patient requirement (`REQ002`). Listing 4.6 presents the set of filtered trace links which satisfy this scenario.

Listing 4.6: Easy Clinic - Filtered Trace Links Example

```
1 tracelink t2 = {REQ002, Dependency, Test Case, "TC02:Register patient with invalid national
      ID number"}
```

Finally, in the step ⑧, it is necessary to decide if extracted trace links satisfy the task which drove their extraction. The process is finished once extracted trace links assist the motivating task. Otherwise, tagged artifacts need refinement and a new iteration is necessary. It is important to highlight that deciding if extracted trace links are satisfactory or not is a challenging task [22, 27], thus this decision is up to project's stakeholders.

Software engineers are also the main actors of the extraction phase. They manipulate the trace links generated in the production phase in order to retrieve requirement related artifacts. Notwithstanding, software engineers should agree with stakeholders and requirement engineers decision about the satisfiability of extracted trace links.

4.2 Process' Contracts

In addition to the process workflow, the definition of actors, activities and responsibilities, it is important to emphasize process contracts and how the different process phases interoperate. The contracts specification was motivated by traceability grand challenges [22], which describe the necessity of defining contracts that provide support for instantiating traceability roles and responsibilities as well as establishing how they exchange information. By providing interfaces to traceability activities, traceability contracts foster portable traceability and provide means by which different organizations can exchange traceable information.

Considering the traceability model presented in Chapter 2, five major contracts were defined for the proposed requirements traceability process (Figure 4.1), which describe that it should be: *(1)* traceable; *(2)* translatable; *(3)* indexable; *(4)* searchable; and *(5)* renderable. The proposed contracts guide trace links creation and manipulation and also enable the portability of the proposed process. They encompass the whole traceability model, defining major services provided by each phase. The tracing, translating and indexing contracts consider the production phase, they provide interfaces to the activities of parsing and translating tagged artifacts to the proposed trace link representation as well as grouping and indexing trace links. On the other hand, the searching and rendering contracts encompass the extraction phase. They provide interfaces to the activities of querying and filtering trace links as well as rendering the output of such activities. Notice that, for each contract, there is one or more services that comply with the proposed contract. Furthermore, each contract has a set of operations that must also be provided by the implementing service.

The tracing and translating contracts consider the activity of translating tagged artifacts into the adopted representation. In order to achieve so, the **tracing contract**, detailed in Table 4.1, offers services to extract trace links from tagged artifacts. Its major operation, **parse** dictates that the implementing service produces trace links related to one or more types of traced artifacts. Such contract, is closely related to the production of trace links and relies on existing techniques to extract the relationships between requirements and artifacts, detailed in Chapter 2. As an example, EasyClinic test cases are written in TestLink. Thus, a `TestLinkTracingService` would extract the traces of this artifact type. If other types of artifacts are to be traced, other services complying with the tracing contract would be

Table 4.1: Tracing Contract

parse	**pre:** Traceable artifacts are tagged according to the defined strategy **post:** All tagged artifacts are parsed and trace links are produced

Table 4.2: Translating Contract

write	**pre:** Trace links were produced **post:** The set of produced trace links is written to the adopted TRL
read	**pre:** Trace links are represented according to the adopted TRL **post:** The set of produced trace links is read from the adopted TRL

specified.

Once artifacts are parsed, the **translating contract** defines how the set of parsed trace links should be represented according to the described TRL. Table 4.2 presents the translating contract. Its main operation, **write**, define that trace links can be written to the described representation, guaranteeing that they can be created and manipulated. Analogously, the **read** operation defines that traces represented in the adopted representation can be read. Notice that, the tracing and translating contracts are closely related and the parsing and read/writing process usually occur sequentially.

Indexing and searching contracts are closely related. Their design should consider how trace links are stored (either in memory or in disk). The **indexing contract**, detailed in Table 4.3, offers services to communicate with the storage service to group and index trace links. Its four operations detail the addition and removal of trace links from the storage system. The addition can occur in a batch manner, through the **index** operation or singularly, through the **add** operation. Similarly, the **clear** and **remove** operations detail how trace links are removed from the storage system.

Table 4.3: Indexing Contract

clear	**pre:** Storage system is available **post:** All grouped/indexed trace links are removed from the storage system
index	**pre:** Storage system is available **post:** Trace links are grouped/indexed into storage system
add	**pre:** Storage system is available; Trace link does not exist in the storage system **post:** A new trace links is added to the already grouped/indexed trace links
remove	**pre:** Storage system is available; Trace link exists in the storage system **post:** An existing trace links is removed from the grouped/indexed trace links
update	**pre:** Storage system is available; Trace link exists in the storage system **post:** An existing trace links is updated into the grouped/indexed trace links

Table 4.4: Searching Contract

query	**pre:** Storage system is available; Trace links can be queried; **post:** Trace links that conform to passed parameters are retrieved from the storage system
filter	**pre:** Storage system is available; Trace links were queried; **post:** The set of previously retrieved trace links is filtered according to passed parameters

Table 4.5: Rendering Contract

render	**pre:** Existing data structure to represent traceability relationships **post:** Trace links are represented according to the selected data structure

Once trace links are stored, the **searching contract** offers services to communicate with the storage service in order to retrieve them. It considers that a storage service is available and the existence of fields that can be used as a query criteria. Hence, once the **query** operation is executed, all trace links that conform to query parameters should be retrieved. Moreover, the **filter** operation describes that retrieved trace links can be filtered according to a filter criteria. Thus, the query and filter operations encompasses the extraction phase, of the traceability model.

Finally, once trace links are retrieved, it is necessary to render them. In such context, Table 4.5 details the **rendering contract**, which defines how extracted trace links will be displayed. Its major operation, **render** considers the existence of data structures that can be manipulated in order to render and represent trace links in a meaningful way. For instance, graphs, trees or traceability matrices could be considered as possible data structures to organize extracted trace links [43,55]. As a post condition, the contract's operation states that the trace links are rendered according to selected data structure.

4.3 Chapter Debriefings

The benefits of requirements traceability have been acknowledged by different researches both from academy and practice [55, 58] and, as a consequence different maturity levels dictates its use [12]. Even though, there is not a consensus about the major aspects of the requirements traceability process. The heterogeneity of organizational processes preclude that traceability practices can be unified across different organizations and, consequently, that traceability processes can be ported.

Identify means to extract common aspects of the requirements traceability process and promote its portability is one of the grand challenges of requirements traceability [22]. Therefore, we proposed a requirements traceability process which underlies on the traceability model (*i.e.* definition, production and extraction). Moreover, while designing the proposed process, we have considered a common trace link representation and established major contracts that prescribe how the process phases interoperate. Thus, the proposed process can foster the discussion of key aspects related to the traceability process as well as its roles, responsibilities, and contracts.

As a final remark, it is important to emphasize that the proposed process does not encompass all possible scenarios in which requirements traceability can be used [45]. As a consequence, it may not be adequate to some organizational environments or some requirement traceability tasks, *e.g.* requirements rationale [45]. Even though, the proposed process fosters the discussion of major aspects of requiremenst traceability and how portable traceability can be addressed.

Chapter 5

Tool Support

As a first step towards automation, we developed a prototype tool – Service Oriented Requirements Traceability Tool (SORTT)[1] – which automates part of the activities described in the proposed requirements traceability process, presented in Chapter 4. Therefore, in this chapter, we present the tool's overview (Section 5.1), architecture (Section 5.2) and its execution flow (Section 5.3) as well as overall considerations about it (Section 5.4).

5.1 Overview

SORTT is a service oriented requirements traceability tool. It automates the activities of *(1)* producing trace links, *(2)* translating then into the proposed traceability representation language, *(3)* indexing and *(4)* searching them as well as *(5)* rendering retrieved trace links. To this extend, a main module interacts with a series of services through their provided operations. Therefore, one can trace requirement related artifacts through SORTT's functionalities.

Figure 5.1 details SORTT workflow. First, in ①, the set of traceable artifacts is sent to the parser module, which will produce the trace links in ②. The produced trace links are the input of the translator, which translate them according to the adopted trace link representation. Once trace links are translated, in ③ the translator output is indexed in the storage system through the indexer module. Additionally, translated trace links are returned to the main module, hence parsed requirements, artifacts, types of artifacts, and types of relation-

[1] Available at `http://goo.gl/STU3kf`

ships can be visualized and manipulated. Considering the necessity to extract trace links, one must specify queries through the adopted language. Thus, in ④, the specified queries are parsed and sent to the querier, which will execute them, according to declared parameters. Finally, in ⑤ the result of an executed query is returned to the renderer, which displays it according to a defined data structure.

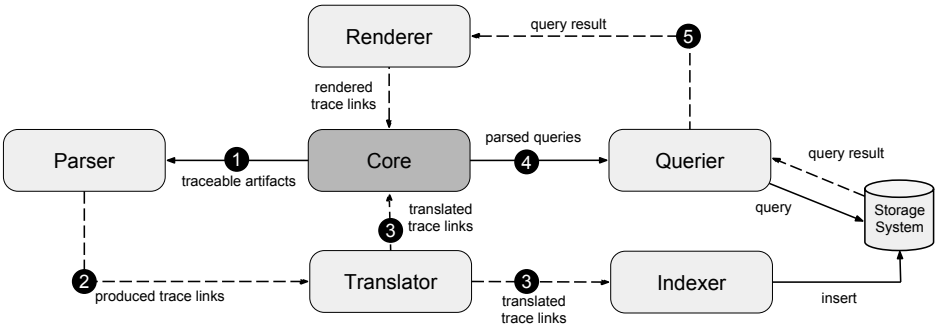


Figure 5.1: SORTT - Overview

Notice that SORTT workflow is closely related to the defined traceability process, presented in Chapter 4. Particularly, it considers the defined traceability contracts as well as the defined trace link representation language as major artifacts, which traverse through SORTT modules/services.

5.2 Architecture

In order to automate the activities of producing, translating, indexing, querying, filtering and rendering trace links, SORTT integrates five major services, which *(1)* produce trace links; *(2)* translate them according to one language's constructions; *(3)* index extracted trace links; *(4)* search and retrieve trace links, according to language's queries; and *(5)* render them, based on a defined visualization data structure. In such context, tool's services work almost independently and are highly customizable. Nevertheless, their underlying communication is based on the proposed requirements traceability contracts (Chapter 4).

SORTT's architecture is built upon a service oriented architecture (SOA), which considers the traceability contracts as its cornerstone. Its architecture comprising modules, services and the contracts that they rely on are presented in Figure 5.2. Its main module, the

extractor integrates the renderer, indexer and querier services. In turn, the parser and translator services produce trace links and communicate with the storage system in order to store them. Regarding the previously mentioned architecture, SORTT is intended to be highly configurable. Services can be (de)attached to its core module and different services can be provided according to organizational needs.

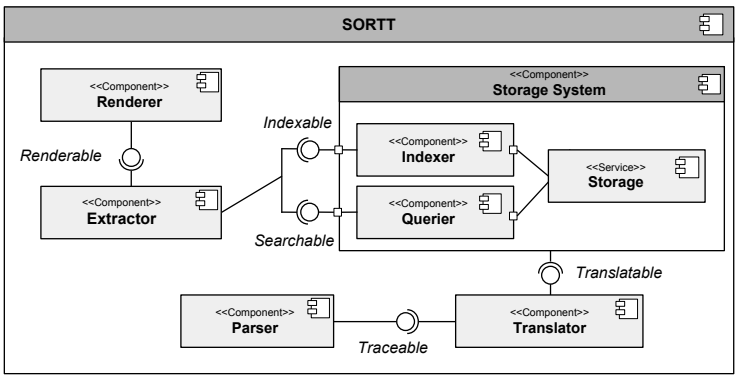


Figure 5.2: SORTT - Architecture Overview

In its current implementation, SORTT has services to extract trace links between requirements and test cases as well as services to translate the already extracted trace links from benchmarks of the Center of Excellence for Software Traceability (CoEST). Its underlying storage system is implemented using Apache Solr[2] and, consequently, the indexer and querier services are implemented through its application programming interface (API). Finally, the renderer service is implemented using the JTree API, thus trace links are displayed in a set of hierarchical trees of nodes.

5.3 Usage

As an example of SORTT's workflow, let us consider a requirement traceability task supported by it. To this extent, let us revisit the traceability process (Figure 5.3) proposed in Chapter 4 and the EasyClinic example. In such example, a requirements engineer is in charge of identifying artifacts related to the booking visit requirement.

Considering the CoEST's EasyClinic application, source code files and test cases are defined as traceable artifacts. Thus, in the step ① of the definition phase (Figure 5.3), these

[2] `http://lucene.apache.org/solr/`

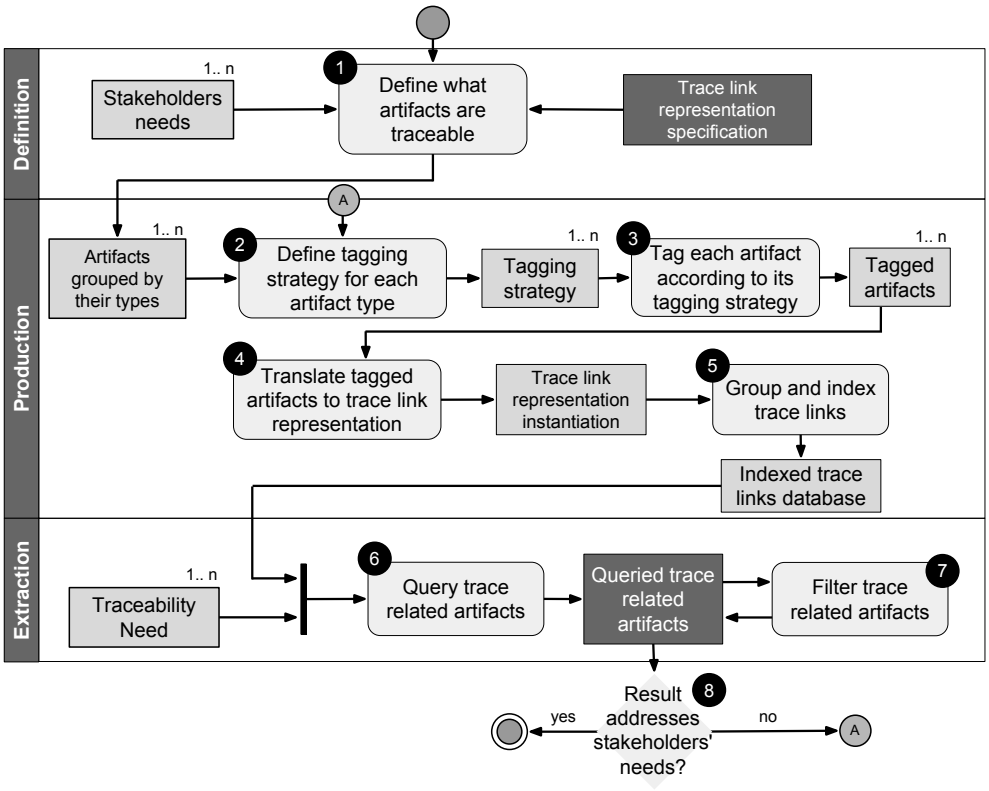


Figure 5.3: Traceability Process

artifacts are defined as traceable. In the production phase (Figure 5.3), in the step ②, a tagging strategy is defined for each artifact type. Therefore, in the step ③, test cases and source code files are tagged according to defined strategies. Notice that, we merely cite steps 1 to 3 for the sake of completeness[3]. Nevertheless, SORTT's workflow encompasses the remaining steps of the proposed process (4 to 8).

In order to translate tagged artifacts (step ④), in SORTT properties file, the engineer sets the directories or archives to be traced, as presented in Listing 5.1. These properties will be the input of the `CoESTParser` and `CoESTTranslator` services, as further detailed.

Listing 5.1: EasyClinic - Requirement Example

```
1 #easyClinic
2 coest.requirements=/coest/easyClinic_Requirements.xml
3 coest.artifacts=/coest/easyClinic_Classes.xml;/coest/easyClinic_TestCases.xml
4 coest.traces=/coest/easyClinic_Links_UC_CC.xml;/coest/easyClinic_Links_UC_TC.xml
```

[3]CoEST's benchmark does not describe how these steps were carried out and it only presents traceable requirements, artifacts, and their relationships as XML files

Once properties are set, the tool is executed. Figure 5.4 presents SORTT initial screen. It displays tabs for traced artifacts, such as test cases, source code files, and use cases. In turn, a core tab displays requirements and declared queries. Regarding the initial screen, it has three main functionalities, which *(i)* produces the trace links; *(ii)* index them; and also *(iii)* query requirement related artifacts.

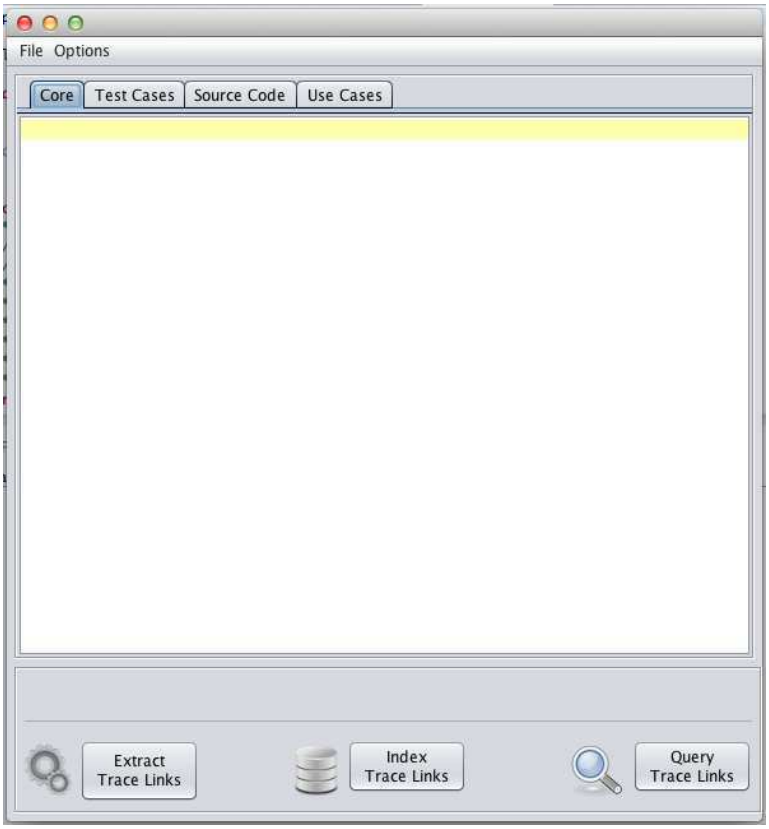


Figure 5.4: SORTT - Main Screen

Considering the step ④ of the production phase, through the `extract trace links` functionality, SORTT communicates with the `CoESTParser` and `CoESTTranslator` services. Therefore, trace links are produced and requirements and artifacts are translated to the TRL's format. As an example, Figure 5.5 presents the tool's output. Each one os its tabs are populated according to language's constructions. For instance, in the `Core` tab, requirements are presented as well as language's queries. On the other hand, the remaining tabs presents traceable artifacts, such as the excerpt of test cases presented in the `Test Cases` tab.

Once trace links are produced, they must be indexed. In order to group and index them, in

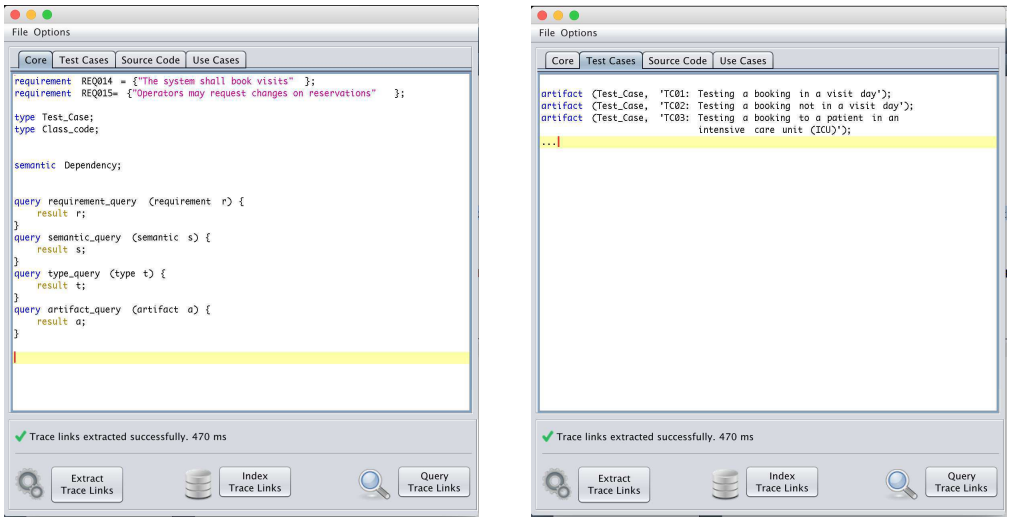


Figure 5.5: SORTT - Core and Test Cases Tabs

the step ⑤ of the production phase (Figure 5.4), the `index trace links` functionality is executed. Listing 5.2 presents an excerpt of the indexer operation output. Considering that the indexer service underlying storage system is implemented using Apache Solr, a reverse index strategy [20] group and index elements by their related requirements.

Following process' execution flow, in the step ⑥ of the extraction phase (Figure 5.4) one must query artifacts related to the booking requirement. In order to trace them, first it is necessary to declare TRL's queries, as the ones presented in Figure 5.5 core tab. The queries will search and retrieve produced trace links according to their declared parameters and result expression. For instance, the `requirement_query` is suitable to search artifacts related to the booking requirement.

Listing 5.2: SORTT - Grouped and Indexed Trace Links

```
1 artifact, semantic, artifact_type, version, id, requirement
2 Booking.java, Dependency, Class_code, 1485318975184175104,23, "REQ014, REQ015"
3 BookingAgenda.java, Dependency,Class_code, 1485318975190466560, 24, "REQ014, REQ015"
4 Patient.java, Dependency, Class_code, 1485318975066734592, 5, "REQ014, REQ015"
5 Visitor.java, Dependency, Class_code, 1485318975214583808, 28, "REQ014, REQ015"
6 ...
```

In order to query requirement related artifacts, the `query trace links` functionality is executed. It parsers the specified queries and send them to the querier, as presented in Figure 5.6. In the query menu, parsed queries are displayed in the up left corner, grouped by their names. In the bottom left corner, selected query's expression body is displayed and its incoming parameters can be edited by changing the tag(s) `<value>`. Once query's

parameters are edited, the query can be run through a request to the querier service using the run query button.

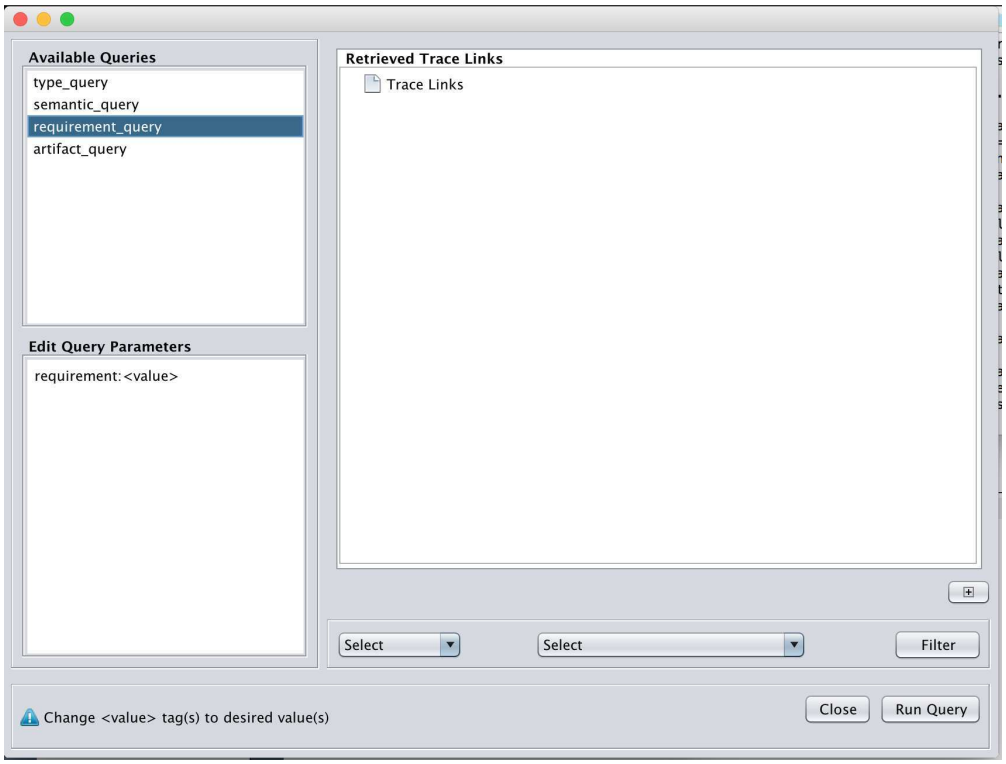


Figure 5.6: SORTT - Query Screen

As an example, Figure 5.7 presents the output of the `requirement_query` passing the booking requirement as a parameter (`REQ014`). Notice that, the result is structured according to the renderer service. Thus, trace links are displayed as a set of hierarchical tree of nodes.

As the result is displayed, the engineer decides to filter the result, visualizing only the test cases related to the booking requirement. Therefore, in the step ⑦ of the extraction phase (Figure 5.4), trace links are filtered through the filter operation. Figure 5.8 present operation's output. Similarly to the query operation, the result is structured according to the renderer service.

Finally, in the step ⑧ of the extraction phase (Figure 5.4), the engineer must decide if the extracted trace links attend to the task which drove their extraction. If extracted trace links are suitable, then the process is finished. Otherwise, a new iteration should be considered.

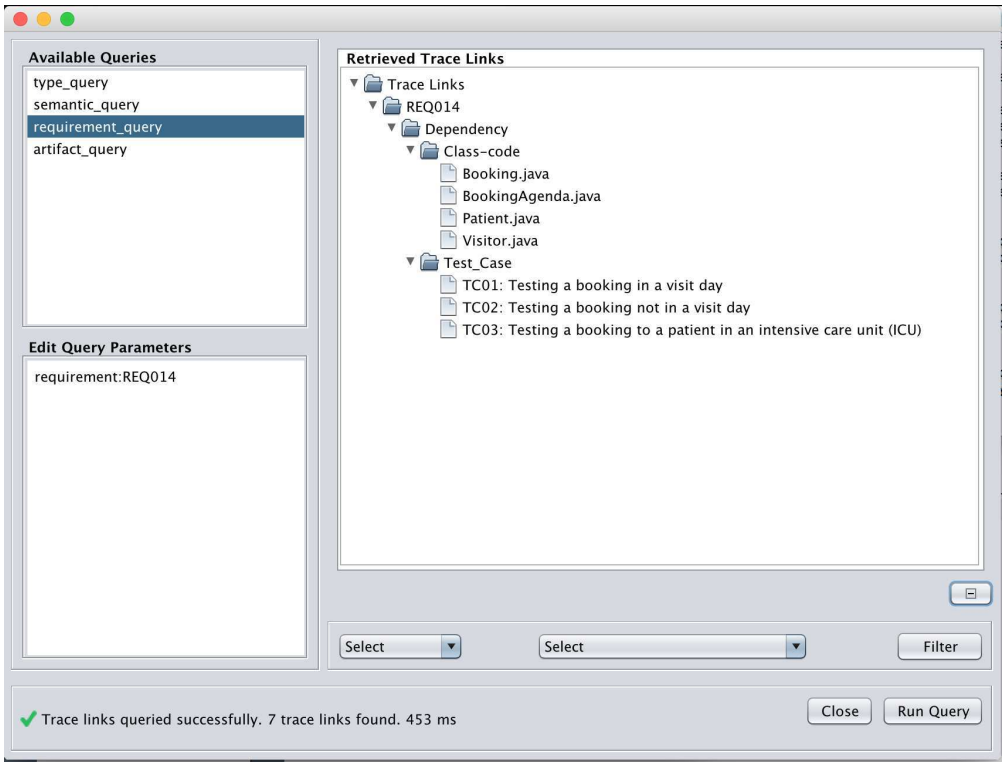


Figure 5.7: SORTT - Query Output

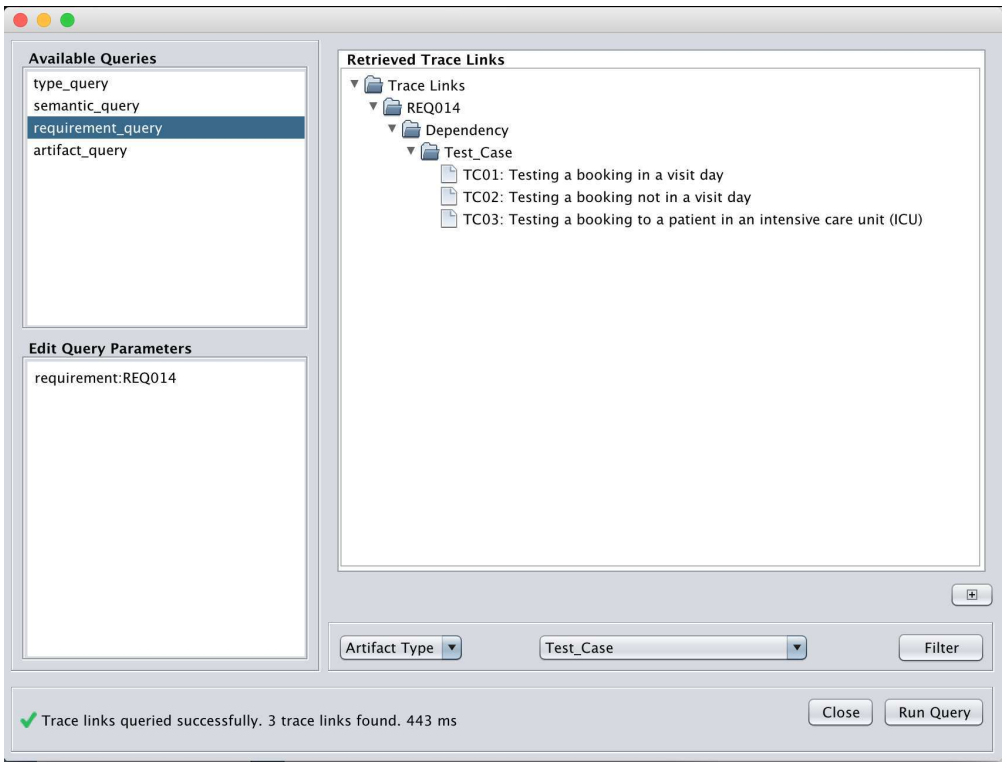


Figure 5.8: SORTT - Filtered Output

5.4 Chapter Debriefings

As observed by Kannenbergnd and Saiedian, the number of traceability links that need to be captured grows exponentially with the size and complexity of the software system. This means that manually capturing traceability data for large systems requires an extreme amount of time and effort [27]. Thus, requirements traceability can be a burdensome, time consuming and elusive task [3, 21, 46].

Considering the time and effort to extract trace links, Gotel and Finkelstein state that an adequate tool support is essential for overcoming requirement traceability burdens [21]. Notwithstanding, computer-aided software engineering tools (CASE tools) do not address the particularities of each organizational need. Therefore, organizations must create their in-house tools [46], which could likely be used in similar contexts if portable and scalable traceability are considered through the tool's design [22]. Otherwise, tools are organizational specific and cannot be ported.

Regarding the aforementioned issues, we designed SORTT, a service oriented requirements traceability tool. Considering portable and scalable traceability, SORTT automates activities of the requirements traceability process through its plugabble services. Their communication rely on established contracts, which dictates how different requirement traceability activities would exchange information. Hence, customization is addressed while maintaining defined interfaces which can be exploited by different organizations.

Chapter 6

Evaluation

In this chapter, we present how the proposed traceability representation language (TRL) and the proposed traceability process were evaluated. Considering that the evaluation was twofold: *(i)* proposed language; and *(ii)* proposed process, we divide the presented evaluation in Sections 6.1 and 6.2. Both evaluations were structured considering a goal, question, metric approach [7]. Therefore, for each evaluation, we present its objective, methodology, results and discussion.

6.1 Language Evaluation

Regarding the proposed language, we elaborated an empirical experiment to evaluate it. The experiment was structured considering the goal, question, metric approach [60]. Therefore, our objective is to **analyze** the proposed language **in order to** evaluate it in comparison with traceability languages[1] **according to** languages' simplicity **in the point of view of** system developers **in the context of** trace links extraction. Therefore, our experiment aims at evaluating the understandability and the ease of use of our proposed language in comparison with existing ones.

In order to understand the conducted experiment, the next subsections further detail its planning (Section 6.1.1), results (Section 6.1.2), and discussion (Section 6.1.3).

[1]The compared languages are classified as traceability query languages. Even tough, they provide support to the declaration of requirements and artifacts and also have an underlying trace link data structure.

6.1.1 Planning

In order to analyze the proposed requirements traceability language, we detail in the following subsections the experiment's objective, hypotheses, object of study, subjects, designed questionnaire, variables, metrics, as well as the overall experiment's design.

Objective

In order to evaluate the proposed TRL, we have considered traceability query languages, present in the literature. To this extent, we compared TRL with TracQL (Traceability Query Language) [57] and also TQL (Trace Query Language) [34]. Thus, the objects of study of the experiment are:

- **TRL** – the proposed language presented in this dissertation and detailed in Chapter 3;
- **TracQL** – a graph-based traceability query language built on Scala. TracQL considers trace links as a graph like data structure, in which artifacts and requirements are uniformly represented as vertices and their links as edges [57];
- **TQL** – a traceability query language built on top of XML, which considers that requirements and artifacts are nodes and trace links are modeled as locators to these nodes [34].

It is important to highlight that the selected traceability query languages, namely TracQL and TQL, focus on query capabilities in order to search and retrieve trace links. Even though, they also provide support to the declaration of requirements and artifacts. Moreover, they also have a data structure to represent their trace links. Additionally, the authors of all traceability query languages used in the designed experiment were contacted through their institutional emails and the conductor explained his work and intended experiment. In such scenario, the authors were asked for possible running tools and extra documentation, which could clarify any gap in the compared languages. Therefore, all languages could be compared without bias.

Regarding selected languages, the experiment compares them according to readability and writability criteria. **Readability** is how well one can read and comprehend the constructions of a given language. On the other hand, **writability** is how well one could write

programs/code in a given language. Therefore, our experiment aims at evaluating the understandability and the ease of use of our proposed language in comparison with existing ones.

Furthermore, despite not being the main objective of the experiment, we also observed how the proposed language contributes towards portable and scalable traceability [22].

Hypotheses

The experiment's major null hypothesis is that there is no difference between languages simplicity or ease of use. Such hypothesis is decomposed into more specific ones, *i.e.* there is no difference between the readability and writability of TRL and TracQL ($H_{\varnothing 1}$ and $H_{\varnothing 2}$) and; there is no difference between the readability and writability of TRL and TQL ($H_{\varnothing 3}$ and $H_{\varnothing 4}$).

$$H_{\varnothing 1} : TRL \text{ readability} = TracQL \text{ readability} \tag{6.1}$$

$$H_{a1.1} : TRL \text{ readability} > TracQL \text{ readability} \tag{6.2}$$

$$H_{a1.2} : TRL \text{ readability} < TracQL \text{ readability} \tag{6.3}$$

$$H_{\varnothing 2} : TRL \text{ writability} = TracQL \text{ writability} \tag{6.4}$$

$$H_{a2.1} : TRL \text{ writability} > TracQL \text{ writability} \tag{6.5}$$

$$H_{a2.2} : TRL \text{ writability} < TracQL \text{ writability} \tag{6.6}$$

$$H_{\varnothing 3} : TRL \text{ readability} = TQL \text{ readability} \tag{6.7}$$

$$H_{a3.1} : TRL \text{ readability} > TQL \text{ readability} \tag{6.8}$$

$$H_{a3.2} : TRL \text{ readability} < TQL \text{ readability} \tag{6.9}$$

$$H_{\varnothing 4} : TRL \text{ writability} = TQL \text{ writability} \tag{6.10}$$

$$H_{a4.1} : TRL \text{ writability} > TQL \text{ writability} \tag{6.11}$$

$$H_{a4.2} : TRL \text{ writability} < TQL \text{ writability} \tag{6.12}$$

Additionally, the experiment's hypotheses also consider language's trace links and queries comprehension. Hence, specific hypotheses assume that *(i)* there is no difference between TRL and TracQL trace links comprehension ($H_{\varnothing 5}$) as well as TRL and TQL ones ($H_{\varnothing 6}$) and; *(ii)* there is no difference between TRL and TracQL queries comprehension ($H_{\varnothing 7}$) as well as TRL and TQL queries ($H_{\varnothing 8}$).

$$H_{\varnothing 5} : TRL \text{ trace link comprehension} = TracQL \text{ trace link comprehension} \tag{6.13}$$

$$H_{a5.1} : TRL \text{ trace link comprehension} > TracQL \text{ trace link comprehension} \tag{6.14}$$

$$H_{a5.2} : TRL \text{ trace link comprehension} < TracQL \text{ trace link comprehension} \tag{6.15}$$

$$H_{\varnothing 6} : TRL \text{ trace links} = TQL \text{ trace links} \tag{6.16}$$

$$H_{a6.1} : TRL \text{ trace links} > TQL \text{ trace links} \tag{6.17}$$

$$H_{a6.2} : TRL \text{ trace links} < TQL \text{ trace links} \tag{6.18}$$

$$H_{\varnothing 7} : TRL \text{ queries} = TracQL \text{ queries} \tag{6.19}$$

$$H_{a7.1} : TRL \text{ queries} > TracQL \text{ queries} \tag{6.20}$$

$$H_{a7.2} : TRL \text{ queries} < TracQL \text{ queries} \tag{6.21}$$

$$H_{\varnothing 8} : TRL \text{ queries} = TQL \text{ queries} \tag{6.22}$$

$$H_{a8.1} : TRL \text{ queries} > TQL \text{ queries} \tag{6.23}$$

$$H_{a8.2} : TRL \text{ queries} < TQL \text{ queries} \tag{6.24}$$

Regarding experiment's hypotheses, if we reject the null hypothesis for either TracQL or TQL, we will further investigate which language has better outcome by testing their alternative hypotheses (H_{a1} and H_{a8}).

Corpus of the Study

As an object of study, the experiment considered four benchmarks extracted from the Center of Excellence for Software Traceability (CoEST) and a real project under development for the Federal Police of Brazil[2]. Therefore, the experiment's tasks were carried out in these data sets.

Tables 6.1 and 6.2 present the experiment's data set. The CoEST's data set compromises the recurrent example used in this dissertation, the EasyClinic, and also the SMOS, eTour and WV_CCHIT projects[3]. On the other hand, the industrial project corresponds to the e-Pol project, which is currently being developed by the Software Practice Laboratory under

[2]Names and major details of this system are omitted due to privacy policies.

[3]Available at `http://www.coest.org/index.php/resources/dat-sets`

Table 6.1: Data Set Overview

System	Description
EasyClinic	Small health care application to manage medical ambulatories
e-Pol	Federal Police's system to support the process and access to information coming from investigations
eTour	Tour guide system
WV_CCHIT	Health information system
SMOS	High school student monitoring system

Table 6.2: Data Set Numbers

System	#Requirements	#Artifacts	#Trace links
easyClinic	30	110	156
e-Pol	21	129	197
eTour	58	116	308
WV_CCHIT	116	1064	587
SMOS	67	100	1044

a development and research agreement between the Federal University of Campina Grande and the Federal Police of Brazil.

The described data sets have been selected due to the fact that they have trace links between a variety of artifacts, such as client requirements, intern requirements, source code files and also test cases. Hence, we have evaluated each language in different contexts. Additionally, while selecting each project, we have considered the project's number of requirements, artifacts and trace links as presented in Table 6.2 and Figure 6.1. In such context, we highlight that due to privacy policies and time limitations, only 30% of the e-Pol project was used in the experiment.

As a final remark, since the extraction of trace links can be a burdensome and time consuming task [3, 21, 27, 46] and this activity is not evaluated in this experiment, the tasks were carried out in three distinct versions of the tool SORTT. Each version was configured such that it could automatically extract the trace links of each project and translate them into the evaluated language. Therefore, participants could read and analyze the traceable artifacts according to each language's construction and also write the necessary queries in order to retrieve the subset of artifacts related to each assigned task.

Participants

The experiment counted on 14 participants, which were equally selected from an industrial project being developed for the Federal Police of Brazil (e-Pol) and also from M.Sc/Ph.D students of the Software Practices Laboratory[4], both held at the Federal University of Campina Grande.

The selection process was carried out through personal invitations, which were later formalized through email. From 22 invitations, 14 participants were available. Thus, the conductor scheduled a time and date which was adequate for each participant's personal agenda.

Regarding selected participants, they were divided into two groups according to their origin, i.e. graduated and undergraduated ones. Hence, each evaluated system was assigned to at least one participant from each group. Moreover, one of the selected participants of each group was assigned to a pilot execution. Thus, their data was discarded from analysis. As their data was discarded, two systems (EasyClinic and eTour) demanded more participants than the other ones in order to balance them through the discarded participants. Nevertheless, we highlight that all the assignment process was randomly executed.

As a final remark, we highlight that all participants were trained in all traceability languages. Thus, we could mitigate a learning bias and have a common basis among them. Experiment's training considered individual presentations for each participant with the con-

[4] `http://labs-br.org/en/splab/`

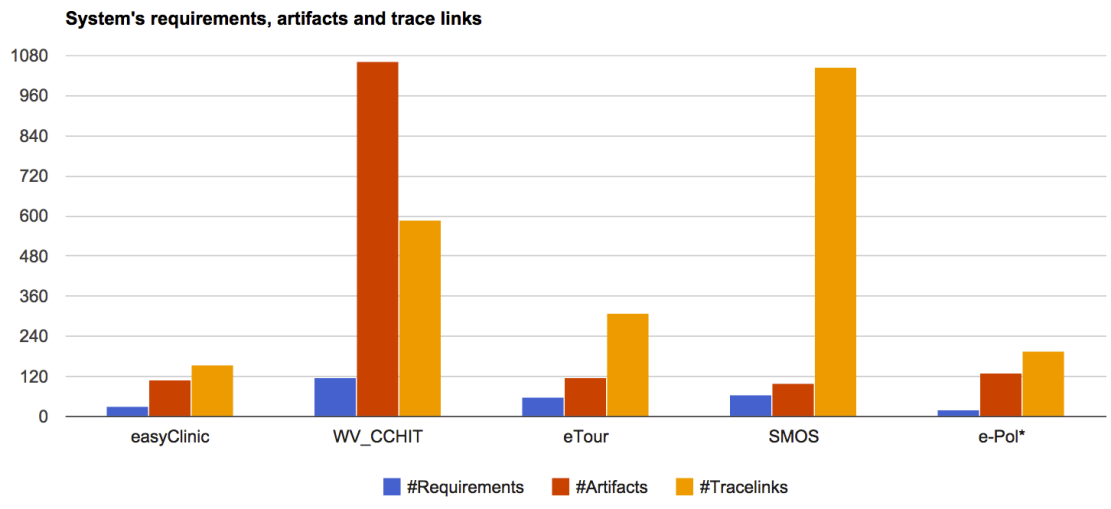


Figure 6.1: Data Set Overview

ductor. Such training detailed each language and their particularities and considering explanations, questions and clarifications, which spent approximately twenty minutes. Once the training was over, all its material was available for later inquiry. It is important to highlight that in order to avoid any bias, the conductor assured that the training was blind, i.e. participants did not known the authors of the presented languages and they were presented impartially.

Tasks

In order to evaluate each language, we have assigned requirements traceability tasks to the selected participants. The tasks were designed considering traceability questions that are likely to arise in the life-cycle of a project, such as the ones discussed by Malletic and Collard [34]. Hence, once a system was assigned to a participant, its tasks were also randomly assigned to him/her.

Table 6.3 presents all the designed traceability tasks groped by their systems. The designed tasks describe the extraction of trace links related to: *(i)* a given requirement; *(ii)* a given requirement, filtering trace links by some artifact type; and to *(iii)* a given requirement, in which the trace links have a specific relation type. Notice that, these tasks have an increasing order of complexity and exercise different elements present in a trace link, such as its requirement, its artifact and also its type of relationship.

Questionnaire

Considering best practices for conducting controlled experiments with human participants [28] and also for comparing domain-specific languages [29], the data for further analysis was gathered through one questionnaire. Therefore, the designed questionnaire is a central artifact in the whole evaluation process and thus, we further detail its conception.

As the first step towards the design of the questionnaire, we surveyed for already existing questionnaires which could be applied in our context. Such research was indeed fruitful and we identified two researches which provided the basis for our questionnaire. The first one, is a family of experiments in order to compare domain specific languages, presented by Kosar *et al.* [29], whereas the second one is a survey applied by Gondim in order to evaluate domain specific languages used in the steps of compiler's constructions [19]. Based on these works,

Table 6.3: Experiment's Tasks

System	Task Description
EasyClinic	Extract all trace links related to the requirement 18
	Extract all trace links from source code artifacts, which are related to the requirement 18
	Extract trace links related to the requirement 10 with a dependency relationship
e-Pol	Extract all trace links related to the requirement UC32
	Extract all trace links from test case artifacts, which are related to the requirement UC12
	Extract trace links related to the requirement UC12 with a dependency relationship
eTour	Extract all trace links related to the requirement UC1
	Extract all trace links from source code artifacts, which are related to the requirement UC1
	Extract trace links related to the requirement UC1 with a dependency relationship
WV_CCHIT	Extract all trace links related to the requirement 1675
	Extract all trace links from requirements artifacts, which are related to the requirement 1677
	Extract trace links related to the requirement 1679 with a dependency relationship
SMOS	Extract all trace links related to the requirement SMOS02
	Extract all trace links from source code artifacts, which are related to the requirement SMOS56
	Extract trace links related to the requirement SMOS56 with a dependency relationship

we further detail the questionnaire's creation.

The designed questionnaire has mostly two types of questions: *(i)* individual evaluation questions; and *(ii)* comparison questions. For each language, in the individual questions, the questionnaire inquires the participants about their comprehension of its constructions and queries. Such questions request that the participant state his comprehension to read or write one language's constructions (either based on the assigned tasks or based on language's code snippets). Then, comparison questions compare each language based on their constructions.

As an example, Figure 6.2 presents one individual evaluation question, based on assigned tasks, whereas Figure 6.3 presents one comparison question.

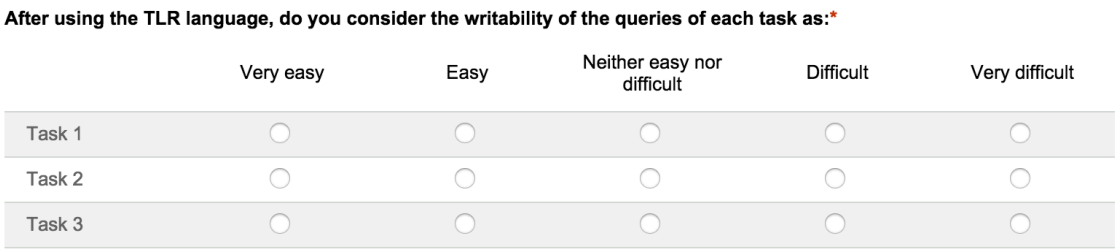


Figure 6.2: Writability Question

Once the questionnaire was elaborated, it was evaluated with a pilot execution, further detailed in this section. Based on it, we could not identify any ambiguous or dubious ques-

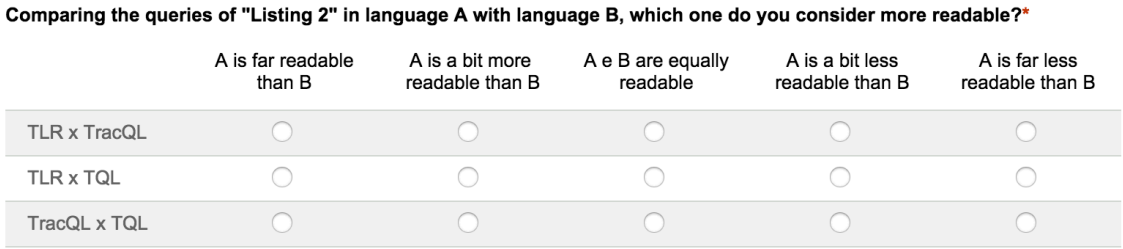


Figure 6.3: Comparison Question

tion. Therefore, the questionnaire was documented, clarifying the purpose of the experiment, specifying a term of consent and also detailing definitions and adopted terms. Hence, the questionnaire was published through the Google survey service and it was sent to the experiment's participants, in the moment of the experiment's execution.

More details of the questionnaire are presented in Appendix A.

Independent and Dependent Variables

Experiment's independent variables covered the selected system, the order of presentation of each evaluated language, participants' experience and also the complexity of the assigned tasks.

Regarding selected languages, assigned systems and participants, independent variables were controlled such that each participant executed the set of previously described tasks in one assigned system for all the three evaluated languages, which were presented in a randomized order. Moreover, assigned systems were balanced, thus being equally distributed among participants. Finally, the participants were blocked in two groups according to their characteristics (system developers or academics).

Experiment's dependent variables considered readability and writability criteria. They were gathered from applied questionnaires, which collected the necessary data for later statistical analysis.

For the languages individual evaluation, a Likert scale measured readability and writability criteria. For instance, Figure 6.4 presents a Likert scale question in which one has to judge the readability/writability of the TRL's requirement construction. In such context, all language individual evaluation questions used the same scale, *i.e.* a range varying from very easy to very difficult.

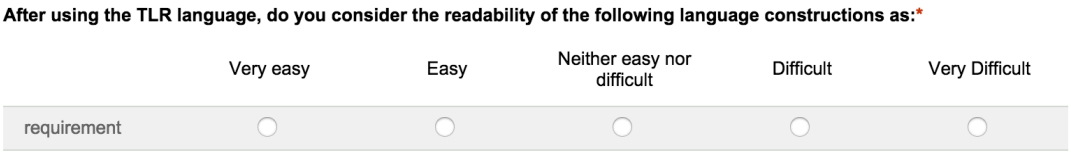


Figure 6.4: Questionnaire Individual Question

Languages pairwise comparison considered comparison matrices, structured using a similar range to the Likert scale questions. Although, in the comparison questions one has to compare all three languages in a pairwise manner. For instance, Figure 6.5 presents a comparison question. In this type of question, one has to compare the left language (referred as `A`) with the right language (referred as `B`). If the left language is considered more readable/writable than the right one, then the question's answer is the leftmost value. Otherwise, it is the rightmost value. In such context, the answers of the comparison questions are converted into a numerical scale, such that: *(i)* equally compared languages have a value of **1**; *(ii)* a slight advantage to one language has a value of **3**; and *(iii)* a significant advantage to one language has a value of **5**. Thus, comparison questions can be further analyzed.

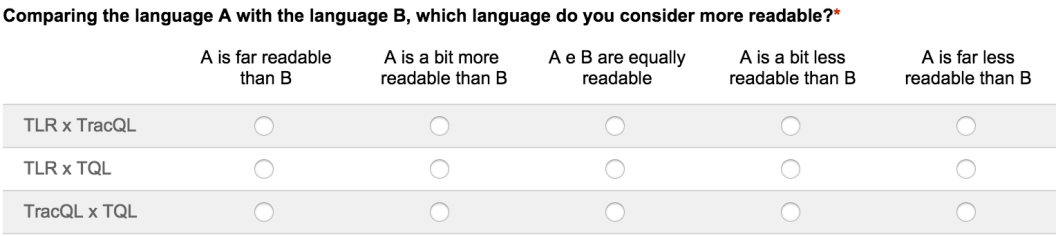


Figure 6.5: Questionnaire Comparison Question

Metrics

Considering experiments dependent variables, by means of our analysis, we have observed *(i)* measures of central tendency for Likert scale questions; and *(ii)* overall comparison ranks, computed according to the assigned comparison's numerical values.

Considering the individual questions, the **mode**, or the most frequent value, was adopted as a measure of central tendency for the analysis of these questions. The mode is normally used for categorical data and is adequate for questionnaires' responses. Therefore, its adoption.

Regarding the comparison questions, their answers are measured according to an **analytical hierarchical process** (AHP). Introduced by Thomas Saaty [50], the analytic hierarchy process is an effective approach for measuring which is the best decision (or choice) by reducing complex decisions in a pairwise comparison. The AHP process can be decomposed in three consecutive steps, such that the approach: *(1)* computes the vector of criteria weights; *(2)* computes the matrix of local option scores; and then *(3)* computes the global scores and ranks the options in decreasing order.

As a concrete example, let us consider one single questionnaire comparison answer. Equation 6.25 presents the first step of the AHP approach while decomposing the languages comparisons. The vector of criteria weights $V = \Big(a_1, a_2, a_3\Big)$ summarizes this answer. TRL has a slight advantage over TracQL (a_1) and a significant advantage over TQL (a_2). Lastly, TracQL has also a slight advantage over TQL (a_3).

$$V = \Big(3, 5, 3\Big) \tag{6.25}$$

Equation 6.26 presents the second step of the approach. The ratio matrix `A` is constructed according to the described vector `V`, such that its upper diagonal values represent the comparison criteria, while the lower diagonal values are its inverse values.

$$A = \begin{pmatrix} & TRL & TracQL & TQL \\ TRL & 1 & 3 & 5 \\ TracQL & \frac{1}{3} & 1 & 3 \\ TQL & \frac{1}{5} & \frac{1}{3} & 1 \end{pmatrix} \tag{6.26}$$

Finally, Equation 6.27 presents the normalized local scores. Such scores are computed by normalizing the ratio matrix and summing each one of its rows, *i.e.* it divides each weight by the sum of the weights in the same column, and then it averages the entries on each row, thus obtaining the score vectors `S`.

$$S = \begin{pmatrix} & TRL & TracQL & TQL \\ TRL & 1 & 3 & 5 \\ TracQL & 0.33 & 1 & 3 \\ TQL & 0.20 & 0.33 & 1 \\ Sum & 1.53 & 4.33 & 9 \end{pmatrix} \rightarrow \begin{pmatrix} - & TRL & TracQL & TQL \\ TRL & 0.65 & 0.69 & 0.55 \\ TracQL & 0.21 & 0.23 & 0.33 \\ TQL & 0.14 & 0.08 & 0.12 \\ Sum & 1 & 1 & 1 \end{pmatrix} = \begin{pmatrix} Avarage \\ 0.63 \\ 0.25 \\ 0.12 \end{pmatrix} \tag{6.27}$$

As a conclusion, in the final step of the approach, one can rank that the TRL language is selected in 63% of the cases, the TracQL one in 25% of the cases and finally, the TQL language in 12% of the them.

Regarding the comparison of the three evaluated languages, the AHP approach can decompose the decision of identifying a better language according to readability and writability criteria in a pairwise manner. Furthermore, such approach provides means to statistically evaluate and rank our data. Therefore, its adoption.

Setup and Procedures

The experiment's setup considered the design of each traceability task according to the particularities of each project. The traceable artifacts of each project were divided into directories and the experiment's tool support was configured such that the trace links were extracted from the traced artifacts according to each language's construction.

Considering that TracQL an TQL languages were evaluated in a proof of concept prototype tool, the language's authors stated that it was not adequate for our planned experiment. Therefore, we configured our own tool support's text editor such that the extracted trace links could be translated to each language's construction, i.e. three different translator services were attached to SORTT according to the evaluated language. This setup was planned in order to provide a uniform environment, in which participants could explore each language equally.

Experiment's procedure followed a defined guide, in which participants were introduced to the three languages used during the experiment, then the conductor provided a small script detailing the overall experiment's structure and after any questions or doubts were clarified the project and its tasks were assigned to the participant. Finally, participants used each language in a randomized order and, after being presented to all languages, answered the questionnaire.

Regarding experiment's setup and procedure, we highlight that a pilot execution was carried out with two participants, one from each group (which were later discarded from the analysis of the results). During the pilot execution participants provided insightful feedback about the experiment's training. They asked specific questions about the languages' syntax and how some queries could be specified. Therefore, the experiment training (examples and

explanations) were further refined in order to better clarify each language's constructions. Moreover, the conductor questioned the participants of the pilot execution about the clarity of the experiment's questionnaire and they stated that its questions were clear and easy to understand. Finally, SORTT's functionalities were exploited without any difficulties and the participants stated that its language highlight was a remarkable feature. Therefore, the pilot execution did not identify any critical point in the experiment and assured that the planned setup was adequate for the experiment's context.

Design

In order to evaluate the proposed language, the experiment followed a **completely randomized design**. Figure 6.6 presents the general overview of the experiment, which considered best practices for conducting controlled experiments with human participants [28] and also for comparing domain-specific languages [29]. First, graduated and undergraduate students from the Federal University of Campina Grande were selected as experiment's participants and then, different traceability benchmarks and an industrial project were adopted as the *corpus* of the study. Considering the selected participants and the *corpus* of the study, we assigned three traceability tasks for each participant in one randomized selected project. The tasks were carried out using the SORTT tool, configured with all evaluated languages, *i.e.* TRL, TracQL, and TQL. Thereafter, a questionnaire was applied to the participants and, thus, the necessary data was gathered and results analyzed.

6.1.2 Results

Once the experiment was run and the questionnaires were applied, the necessary data was collected and then, the results were analyzed.

Regarding the evaluation of the answers of the individual questions, Figures 6.7, 6.8 and 6.9 summarize the questionnaire's responses. In such context, Figure 6.7 summarizes the responses of the TRL language proposed in this dissertation. According to it, it is possible to state the in general participants judged that the language is very easy to read and also very easy to write. Figure 6.8 presents the results of the TracQL language, in which participants stated that it is very easy to read and write. Finally, Figure 6.9 presents the results of the TQL

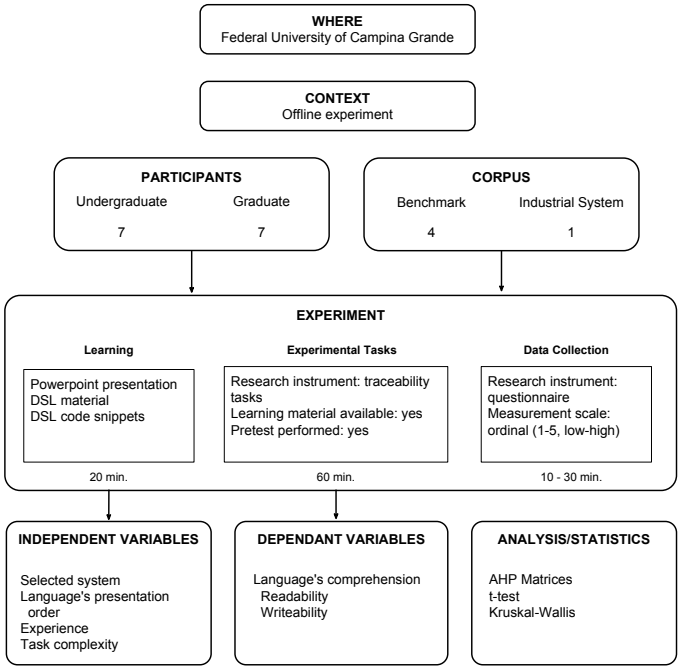
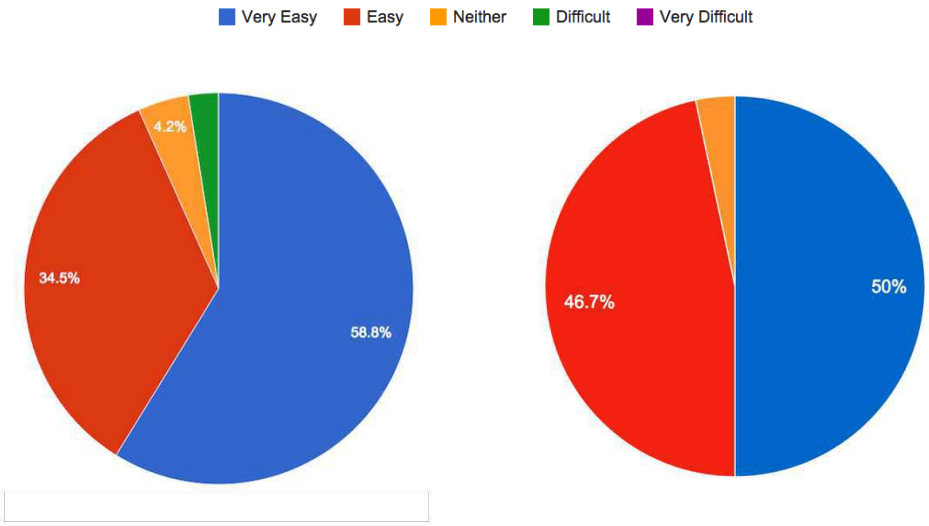


Figure 6.6: Language Evaluation - Experiment's Overview

language, in which the most frequent value also states that participants judged the language as very easy to read and write. In addition to the pie charts with the distribution of responses, Table 6.4 summarizes the most frequent value regarding the readability and writability of each language.

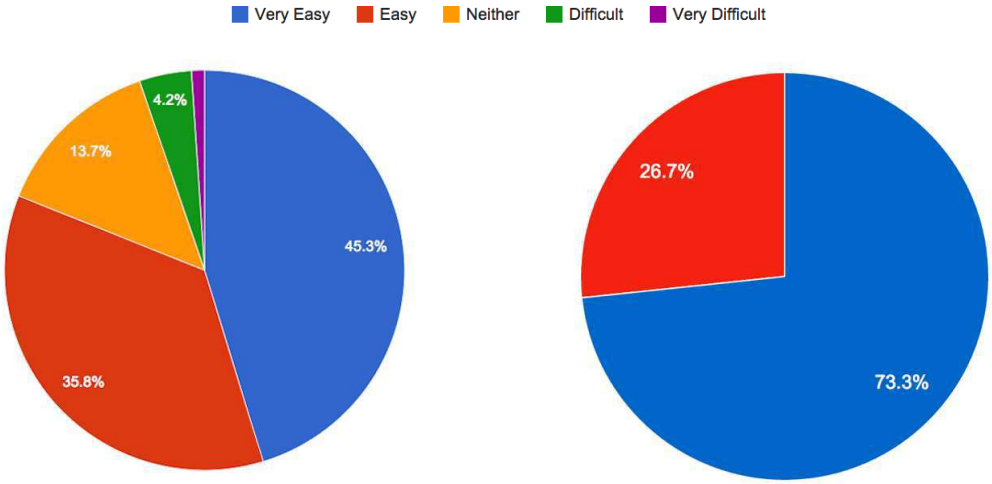
Considering the preceding results, a Kruskal-Wallis statistical test [60] tested experiment's hypotheses $H_{\varnothing 1}$ to $H_{\varnothing 4}$. At a significance level of 5%, test results could not reject the readability null hypotheses ($H_{\varnothing 1}$ and $H_{\varnothing 2}$). On the other hand, writability null hypotheses ($H_{\varnothing 3}$ and $H_{\varnothing 4}$) were rejected and there is statistical difference between languages' writability. By analyzing the computed percentages of each language, presented in Figures 6.7, 6.8 and 6.9, one can rank that TracQL is more writable than TRL and also TQL, whereas TRL is more writable than TQL.

By comparing each evaluated language in a pairwise manner, language's comprehension and particularities can be further differentiated. Hence, we also analyzed the answers of the pairwise comparison questions, through the AHP approach. In these type of questions, participants judged the readability and writability criteria while comparing two queries, written in the three evaluated languages. Moreover, one specific comparison question also requested



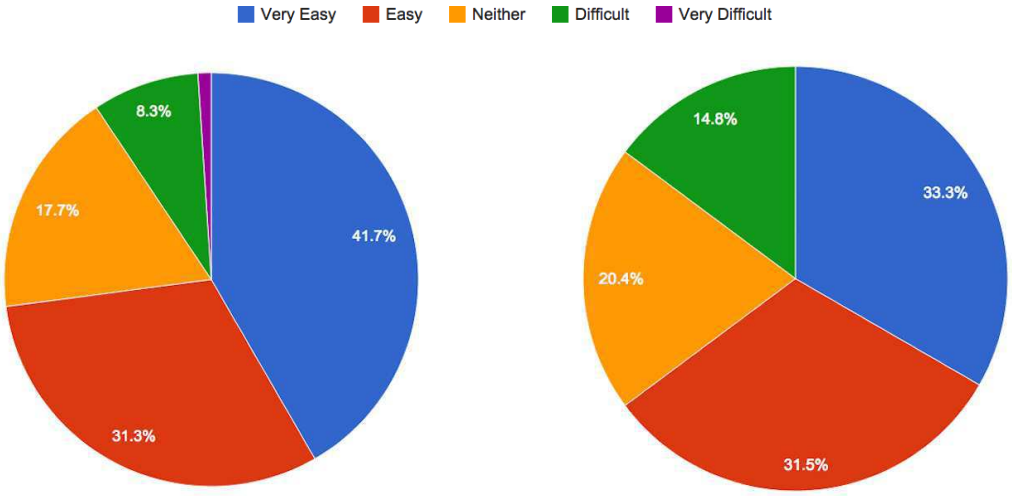
1. Readability **2.** Writability

Figure 6.7: TRL - Questionnaire's Answers Overview



1. Readability **2.** Writability

Figure 6.8: TracQL - Questionnaire's Answers Overview



1. Readability **2.** Writability

Figure 6.9: TQL - Questionnaire's Answers Overview

Table 6.4: Language Individual Question Results

Language	Readability	Writability
TRL	Very easy	Very easy
TracQL	Very easy	Very easy
TQL	Very easy	Very easy

that participants decided which trace link representation was more comprehensible. Therefore, we evaluated experiment's remaining hypotheses, $H_{\varnothing 5}$ to $H_{\varnothing 8}$.

Table 6.5 summarizes all participants' responses to the comparison of the language's trace link representation. According to it, 61% of the responses prioritized the TRL representation, 25% the TracQL representation, and 14% the TQL one. Such results are also supported by t-tests, which confirmed with a significance level of 5% that, for the comparison of TRL and TracQL and also for the comparison of TRL and TQL, the TRL trace link representation was the mostly chosen one. Thus, experiment's trace links null hypotheses were rejected ($H_{\varnothing 5}$ and $H_{\varnothing 6}$), and their alternative hypotheses ($H_{a5.1}$ and $H_{a6.1}$) were confirmed.

Table 6.6 summarizes the comparison of the language's queries. The TRL queries were prioritized in 45% of the cases, whereas the the TracQL ones in 35% of them, and in 20% the TQL queries were prioritized. Considering the small difference between the prioritization of TRL and TracQL queries, we do not have statistical data to state that one of them is mostly adopted (not rejecting $H_{\varnothing 7}$). On the other hand, t-tests confirmed with a significance level of

Table 6.5: AHP Trace Link Representation Result

	P_3	P_4	P_5	P_6	P_7	P_8	P_9	P_{10}	P_{11}	P_{12}	P_{13}	P_{14}	**Rank**
TRL	0.69	0.71	0.66	0.30	0.69	0.55	0.66	0.69	0.69	0.69	0.30	0.69	**0.61**
TracQL	0.21	0.14	0.09	0.61	0.10	0.33	0.09	0.21	0.21	0.21	0.61	0.21	**0.25**
TQL	0.10	0.14	0.25	0.09	0.21	0.12	0.25	0.10	0.10	0.10	0.09	0.10	**0.14**

Table 6.6: AHP Query Results

	P_3	P_4	P_5	P_6	P_7	P_8	P_9	P_{10}	P_{11}	P_{12}	P_{13}	P_{14}	**Rank**
TRL	0.12	0.63	0.61	0.30	0.43	0.61	0.20	0.57	0.61	0.30	0.61	0.14	**0.45**
	0.12	0.66	0.61	0.30	0.43	0.61	0.14	0.57	0.61	0.69	0.61	0.30	
TracQL	0.60	0.11	0.09	0.61	0.14	0.30	0.60	0.29	0.30	0.09	0.30	0.57	**0.35**
	0.60	0.25	0.09	0.61	0.14	0.30	0.57	0.29	0.30	0.21	0.30	0.61	
TQL	0.28	0.26	0.30	0.09	0.43	0.09	0.20	0.14	0.09	0.61	0.09	0.29	**0.21**
	0.28	0.09	0.30	0.09	0.43	0.09	0.29	0.14	0.09	0.10	0.09	0.09	

5% that the TRL queries are prioritized in comparison with the TQL queries (rejecting $H_{\varnothing 8}$ and confirming $H_{a8.1}$).

As a final observation, the questionnaire gave the participants the opportunity to discuss their personal thoughts about either the experiment or the evaluated languages. This was an optional question, which three participants answered as follows:

- *"In the TRL code snippets, I think that the query's call should be present along with its declaration"*;
- *"In general the TracQL language was easier due to the fact that I can mentally visualize its graph structure. Therefore, it was easier to understand its elements and write its queries"*;
- *"In my opinion, in the TRL language, I was more comfortable in reading and writing its constructions and queries"*.

More details of the results are presented in Appendix A.

6.1.3 Discussion

Considering experiment's results, all traceability languages are easily read and written. Nevertheless, while analyzing the experiment's design, its construction and later its execution,

we observed that the languages trace link representation highly influences how artifacts are declared and how queries are constructed.

As presented in Table 6.5, the TRL trace link representation[5] was easily comprehended by participants, hence it was better ranked among all analyzed languages representations. Notice that, since each language's trace link representation influences its query declarations, we also observed the language's query ranking. Table 6.6 presents the overall query ranking among the three evaluated languages. Regarding such rank, the TRL representation was prioritized in 61% of the cases. Therefore, it is possible to state that in general the TRL language presents better results in comparison with the TQL language. On the other hand, TRL and TracQL are fairly equal. Considering that TracQL is, in general, more writable, though TRL presents a more comprehensible trace link representation, both languages are suitable abstractions to requirements traceability. In such context, it is important to highlight that there was no statistical evidence to identify better queries between the TRL and the TracQL languages. However, our overall conclusions are based both on the languages individual analysis and also their pairwise comparisons.

In addition to the quantitative analysis, we qualitatively discuss the proposed language and its tool support based on questionnaire's responses. To this end, we highlight that among all the compared languages, TRL was the only one to support the extraction of trace links in an industrial project (e-Pol) through its tool support (SORTT). In such scenario, as stated by languages' authors, both TracQL and TQL do not provide a downloadable and executable tool which could be used in order to extract trace links from this project. Moreover, TRL tool support is implemented considering a service oriented architecture, hence different traceability techniques can be exploited by the tool. For instance, in order to extract trace links from both the industrial project (e-Pol) and from benchmarks (CoEST), two distinct services were implemented and attached to SORTT. Likewise, different services for the extraction of trace links and also for their search, retrieval and filtering could be attached to the tool. Therefore, the tool can be customized according to different organizations' needs.

Regarding language's queries, participants' feedback stated that, even when TRL's ex-

[5] Notwithstanding, it is important to emphasize current restrictions of the adopted representation. Considering that artifacts are always traced to requirements, direct links between two non-requirement artifacts are not currently supported, and future language improvements will consider them.

pression body clearly stated how parameters are manipulated, the `result` keyword seems ambiguous. As the incoming parameters appears in the result expression, some participants were confused how the result set would be extracted. In turn, TracQL provided filters in its query body, which were easily comprehensible for the experiment's tasks. Though, some participants were intrigued how to manipulate such filters in more complex scenarios. Considering such statements, we believe that the differences between complexity and ambiguity are the likely cause of no statistical evidence to identify a better query between TRL and TracQL. On the other hand, TQL queries were the least prioritized ones, in which participants stated that queries parameters were difficult to understand.

Although it is not the goal of our evaluation, we also discuss how the proposed approach contributes towards portable and scalable traceability [22]. Portable traceability addresses how requirements traceability techniques can be used across different projects or even organizations. We contribute towards it by proposing a trace link representation which was prioritized among all evaluated trace link representations. If such data structure is adopted as a standard, different traceability tools can communicate with each other through common data. Furthermore, the provided tool support also enables that one can attach different services for extracting, indexing, searching and filtering trace links. In such scenario, even when different services are in use, they are abstracted through TRL constructions. Contrarily, neither TracQL nor TQL exploited how they contribute towards portable traceability.

Scalable traceability focuses on inhibiting limits to what type of artifacts can be traceable. In order to address scalable traceability, we first considered a variety of systems with different natures and types of traceable artifacts. Considering such systems, all their requirements, artifacts and trace links could be represented by TRL. Furthermore, since the variety of traced artifacts is likely to increase, new extraction services can be developed and attached to SORTT without compromising a whole traceability process. Hence, our proposed approach contributes towards scalable traceability. In contrast, TracQL exploits only graph abstractions, thus scalable traceability is limited by graph traversing approaches. Finally, TQL illustrated high-level traceability queries without a concrete case study and, as a consequence, scalable traceability is not directly addressed.

Considering experiment results and presented discussion, it is possible to state that the TRL language, and its tool support, provides a feasible abstraction to requirements traceabil-

ity. It is very easily readable/writable, its trace link representation is comprehensible and its queries can retrieve different sets of traces, according to the task at hand.

Threats to Validity

As a final remark, it is also important to mention the threats that we identified to the validity of the experiment as well as how we addressed or mitigated them.

In order to mitigate a **construction threat**, we contacted languages' authors and gathered all the necessary information for the experiment execution. Moreover, we also conducted a pilot execution of the experiment. Therefore, construction threats could be mitigated.

We minimized participants' history and maturation **internal threats** randomizing assigned languages, system and tasks. By controlling/randomizing such factors, we could avoid a learning bias which could favor one specific language.

Finally, due to project's particularities, our obtained results cannot be externalized to other contexts. Notwithstanding, five different systems with different contexts/types of trace links were considered, thus we could mitigate an **external threat**.

6.2 Process Evaluation

Regarding the proposed process and developed tool support, we elaborated an empirical experiment to evaluate them. The experiment was structured considering the goal, question, metric approach [60] and its objective is to **analyze** the proposed requirements traceability process **in order to** evaluate it in comparison with an ad hoc process **according to** effectiveness and performance metrics **in the point of view of** system developers **in the context of** trace links extraction.

In order to understand the conducted experiment, the next subsections further detail its planning (Section 6.2.1), results (Section 6.2.2), and discussion (Section 6.2.3).

6.2.1 Planning

In order to analyze the requirements traceability process, we detail experiment's objective, object of study, subjects, experimental units, variables, metrics, questions and hypotheses, as well as the overall experiment's design.

Objective

The experiment's objective is to analyze the proposed requirements traceability process. To this extent, we compare two requirement traceability processes. The first one is an **ad hoc process**. The second, is the **proposed process** described in Chapter 4. For such comparison, a set of requirements and test cases was presented to participants, and then, they identified the trace links between them.

It is important to highlight that the ad hoc process was selected as a baseline for the experiment because, to the best of our knowledge, requirement traceability processes described in the literature are superficial and do not explicitly describe process phases and activities, precluding that they can be faithfully reproduced.

Hypotheses

Regarding described processes, their effectiveness and performance were measured based on four hypotheses, considering time, precision, recall and efficiency metrics, which are further detailed in this section.

The experiment's major null hypotheses are detailed in Equations 6.28, 6.31, 6.34, and 6.37. The first assumption ($H_{\varnothing 1}$) is that the time spent extracting trace links is equal among both processes. The second assumption is that the precision ($H_{\varnothing 2}$) of both processes is equal, whereas the third hypothesis is that the recall ($H_{\varnothing 3}$) of both processes is equal. Finally, experiment's last assumption ($H_{\varnothing 4}$) is that the efficiency of both processes is equal. If we reject any of these hypotheses, we will further investigate which one has a better outcome by testing their alternative hypotheses (H_{a1} to H_{a4}).

$$H_{\varnothing 1} : T(\text{ad hoc}) = T(\text{proposed process}) \tag{6.28}$$

$$H_{a1.1} : T(\text{ad hoc}) > T(\text{proposed process}) \tag{6.29}$$

$$H_{a1.2} : T(\text{ad hoc}) < T(\text{proposed process}) \tag{6.30}$$

$$H_{\varnothing 2} : P(\text{ad hoc}) = P(\text{proposed process}) \tag{6.31}$$

$$H_{a2.1} : P(\text{ad hoc}) > P(\text{proposed process}) \tag{6.32}$$

$$H_{a2.2} : P(\text{ad hoc}) < P(\text{proposed process}) \tag{6.33}$$

$$H_{\varnothing 3} : R(\text{ad hoc}) = R(\text{proposed process}) \tag{6.34}$$

$$H_{a3.1} : R(\text{ad hoc}) > R(\text{proposed process}) \tag{6.35}$$

$$H_{a3.2} : R(\text{ad hoc}) < R(\text{proposed process}) \tag{6.36}$$

$$H_{\varnothing 4} : E(\text{ad hoc}) = E(\text{proposed process}) \tag{6.37}$$

$$H_{a4.1} : E(\text{ad hoc}) > E(\text{proposed process}) \tag{6.38}$$

$$H_{a4.2} : E(\text{ad hoc}) < E(\text{proposed process}) \tag{6.39}$$

Corpus of the Study

As an object of study, the experiment considered a real project under development for the Federal Police of Brazil, namely e-Pol, previously presented in Section 6.1.1[6].

[6] It is important to emphasize the observed difference in the corpus of the study between the first experiment (Section 6.1) and the second one (Section 6.2). Both experiments considered the e-Pol project [36, 37]. However, due to later revisions and feedback, the proposed language was evaluated with a larger data set [39].

Table 6.7: Data Set Overview

System	Description	#Requirements	#Artifacts	#Trace links
e-Pol	Federal Police's system to support the process and access to information coming from investigations	21	129	197

The studied system supports the processing of information coming from police investigations. Its first internal release, considered in this experiment, has more than 70 use cases and approximately 60 KLOC, divided into more than 30 packages, 600 classes and 400 test cases.

Regarding the aforementioned system, the experiment counted on a subset of its requirements and test cases representing 30% of the overall project's requirements and test cases. This selection was necessary in order to execute the experiment in an affordable time without compromising either participant's schedule or project's deadlines. Table 6.7 presents experiment's data set overview, which randomly selected 21 requirements, with 129 related test cases, and 197 trace links.

As a manner of comparison, an oracle containing the trace links related to the corpus of the study was created. In order to construct it, all artifacts related to the extracted subset were manually analyzed and evaluated by the conductor of the experiment, which produced its trace links. Thereafter, they were revised and confirmed by the project's manager.

Participants

The experiment counted on 12 participants, which are related to the e-Pol project. Considering project's privacy policies, we could not select participants freely. Thus, our set of participants was limited to the ones who had a non disclosure agreement with the project.

Considering our set of participants, we applied questionnaires in order to identify participants' knowledge of the evaluated system and requirements engineering practices. Therefore, we divided participants into novices or experienced ones. Novice participants are newcomers, which had just begin working in the project and did not had coursed the software engineering discipline. On the other hand, experienced participants had at least one year of experience in the project and had coursed the software engineering discipline.

The selection process was straightforward. We asked project's manager about participant's availability and then, they scheduled an affordable time and date to participate in the

experiment.

Tasks

In order to evaluate each process, we have assigned requirements traceability tasks to the selected participants. They require that all existing trace links related to one requirement should be identified. Based on the assigned process, the tasks were carried out in distinct ways.

In the **ad hoc process**, participants manually identified which artifacts were related to a given requirement. In such context, each participant followed his own approach while identifying requirement related artifacts. For instance, some participants followed a pragmatic approach by analyzing each step of each test case, whereas other participants searched for specific words that could identify a requirement.

On the other hand, the **proposed process** follows previously discussed process phases, detailed in Chapter 4. The set of artifacts that were to be traced was manually analyzed, and the agreed tagging strategy, which was defined by one of the experienced participants, considered explicit keywords to identify requirement related artifacts. Thus, this set of artifacts was tagged by the participants according to the defined strategy. Thereafter, the remaining phases were supported by SORTT, which extracted and indexed trace links from tagged artifacts. Later, participants exploited SORTT's query mechanism and searched for requirement related artifacts.

Independent and Dependent Variables

Experiment's independent, variables considered the assigned process and also the assigned requirement. Furthermore, we considered the participant's experience as a noisy variable which could interfere on our data analysis. On the other hand, the dependent variables are the time spent on executing the whole process and the set of extracted trace links.

Metrics

Considering experiments dependent variables, in order to measure process performance and effectiveness, by means of our analysis, we have observed the following metrics:

- **Total time** *(T)* spent per process. For the ad hoc process, measuring the the total time is straightforward, since there are no distinct phases and the participants followed their own steps. However, for the proposed process, the total time is measured by the sum of each process phase (*i.e.* definition, production, extraction), whether manual or automated.

$$T_{\text{ad hoc}} = T_{\text{execution}} \tag{6.40}$$

$$T_{\text{proposed process}} = T_{\text{definition}} + T_{\text{production}} + T_{\text{extraction}} \tag{6.41}$$

- **Precision** *(P)* is the ratio between the number of correctly extracted trace links and the total number of extracted trace links (correct or not). A correctly extracted trace link is one that it exists in the built oracle.

$$P = \frac{N_{\text{corretly extracted trace links}}}{N_{\text{extracted trace links}}} \tag{6.42}$$

- **Recall** *(R)* is the ration between the number of correctly extracted trace links and the total number of correctly existing trace links. Existing trace links were computed in the experiment's oracle.

$$R = \frac{N_{\text{corretly extracted trace links}}}{N_{\text{correctly existing trace links}}} \tag{6.43}$$

- **Efficiency** *(E)* is the ration between the harmonic mean of precision and recall (F-measure [48]) and the total time spent to execute the process. The F-measure can be interpreted as a weighted average of the precision and recall, where its best value is 1 and worst 0.

$$\text{F-measure} = 2\frac{P.R}{P+R} \tag{6.44}$$

$$E = \frac{\text{F-measure}}{T} \tag{6.45}$$

Regarding the aforementioned metrics, they are observed considering the **median** as a measure of central tendency. The median is the middle score for a set of data. It is less susceptible to outliers and skewed data. Therefore, its adoption.

Setup and Procedures

In the setup of the experiment, the conductor gathered copies of experiment's requirements in a directory and also created a TestLink project with the copies of the subset of test cases do be traced. Therefore, we had a controlled environment with all necessary artifacts.

Experiment's procedure followed a guideline. Participants were introduced to major concepts of requirements engineering and requirements traceability and presented to their assigned process and tasks. Thereafter, for each process, participants had an unlimited time to carry out their tasks and at the end of them, they provided their extracted trace links to the conductor, which measured the time spent during their activities.

Design

The experiment followed a **latin square design** [60]. Figure 6.10 presents the general overview of the experiment. Considering participants related to a real industrial project, we extracted the trace links from 30% of the project's requirements and test cases and then, randomly assigned a requirement and a process to a participant, asking him/her to extract test cases in order to relate the assigned requirement to them. Then, we measured the time spent executing each process and the number of extracted trace links, thus the defined metrics were computed according to observed variables. Hence, we could statistically analyze the gathered data and test our hypotheses.

6.2.2 Results

Once the experiment was run and the necessary data was collected, experiment's result were analyzed.

Overall, the experiment spent approximately 232 minutes analyzing and retrieving requirement related artifacts from 129 test cases. The ad hoc process spent 159 minutes with a median of 15 minutes per assigned requirement, whereas the proposed process spent 73

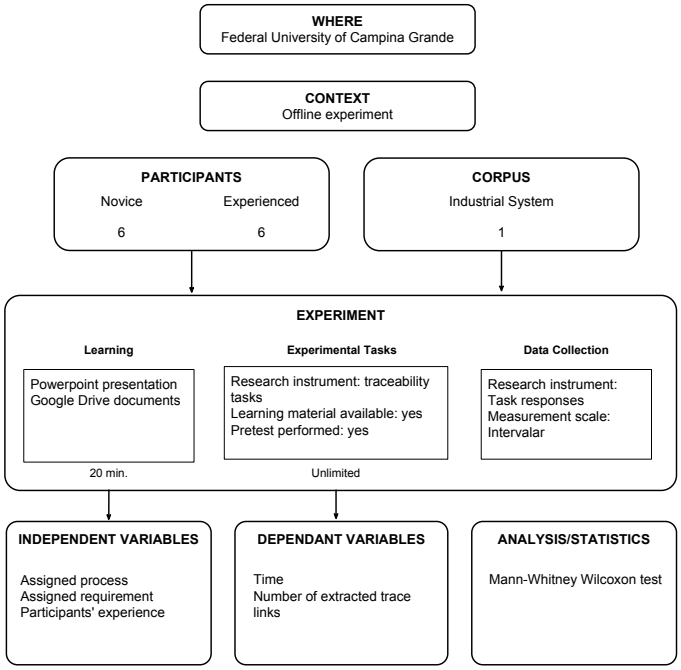


Figure 6.10: Process Evaluation - Experiment's Overview

Table 6.8: Experiment's Results Overview

Metric	Ad hoc	Proposed Process
Time	15 minutes	7 minutes
Precision	100%	100%
Recall	100%	100%
F-measure	100%	100%
Efficiency	6%	14%

minutes, with a median of 7 minutes per assigned requirement. The median of both processes for the precision and recall metrics is 100%. Additionally, the ad hoc process had a median value of 6% for the efficiency metric, while the proposed process had a median value of 14%. Such analysis is summarized in Table 6.8 and Figures 6.11 and 6.12.

Once overall characteristics of our sample were identified, we further investigated our data set in order to perform the hypotheses tests. First, we grouped the results according to participants' experience and observed that there was not a statistical difference between experienced participants and newcomers. Hence, we considered the whole data set of each process and observed that they were not normally distributed. Therefore, we tested our previous defined hypotheses considering a Mann-Whitney Wilcoxon's test and a significance level of 5%.

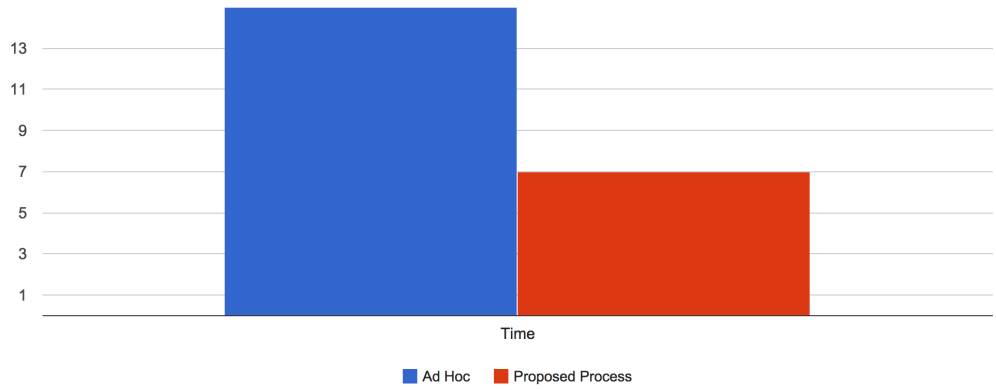


Figure 6.11: Process Evaluation - Performance Bar Chart Comparison

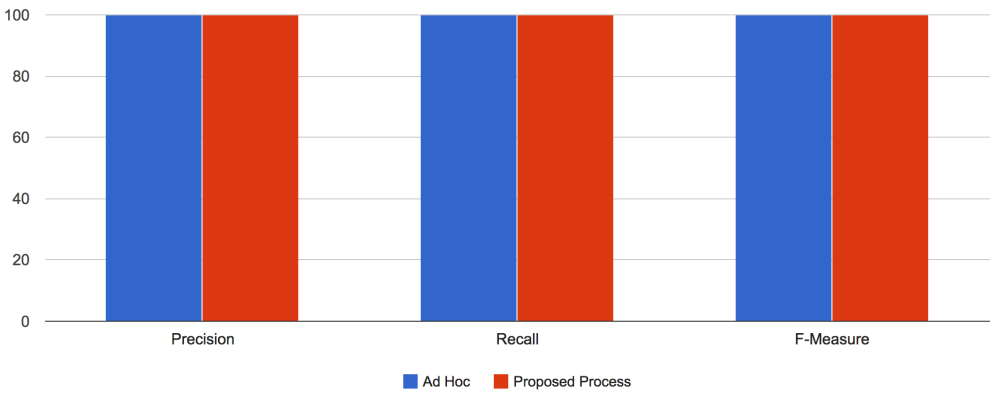


Figure 6.12: Process Evaluation - Effectiveness Bar Chart Comparison

First, experiment's precision and recall hypotheses ($H_{\varnothing 2}$ and $H_{\varnothing 3}$) were tested and neither the hypothesis that the precision of both processes is equal ($\rho = 0.28$) nor the hypothesis that the recall of both processes is equal ($\rho = 0.17$) could be rejected.

Then, the time hypothesis ($H_{\varnothing 1}$) was tested and its null hypothesis rejected ($\rho = 0.0005$). Also, the alternative hypothesis ($H_{a1.1}$) that the time spent on the ad hoc process was greater than the time spent on the proposed process was rejected ($\rho = 0.0002$).

Finally, experiment's efficiency hypothesis ($H_{\varnothing 4}$) was tested. First, the hypothesis that both processes' efficiency are equal ($\rho = 0.0006$) was rejected and then, a second test rejected the alternative hypothesis ($H_{a4.1}$) that the ad hoc process has a greater efficiency ($\rho = 0.0003$).

According to the observed results and hypotheses tests, it is possible to state that the proposed process has a greater efficiency and spends less time executing the requirements traceability process. On the other hand, we do not have statistical evidence to extract conclusions based on the precision and recall metrics.

Experiment's raw results are detailed in the Appendix B.

6.2.3 Discussion

Analyzing experiment results and hypotheses tests, it is possible to state that the proposed process has a better performance ($H_{\varnothing 1}$ and $H_{\varnothing 4}$) in comparison with the ad hoc process. In such scenario, the provided tool support was essential. SORTT decreased the effort to extract trace links by automating process' activities, whereas the ad hoc process pragmatically analyzed all traced artifacts. Additionally, both processes had the same accuracy ($H_{\varnothing 2}$ and $H_{\varnothing 3}$). Most likely, ad hoc tests had 100% accuracy due to the fact that all the set of traced artifacts was analyzed. On the other hand, the proposed process had 100% accuracy as a result of tagging all the set of traced artifacts. Based on such observations, it is possible to state that the proposed process has a better performance in comparison with the ad hoc process. However, such gains are mostly related to the provided tool support and we highlight that benefits provided by it need to be distinguished from the ones provided by the proposed process. For instance, while evaluating processes execution, we emphasize roles and responsibilities. In the ad hoc process, there was not a clear division of roles and responsibilities, thus the effort to extract traces was associated with only one participant. Hence, the ad hoc

execution was burdensome. On the other hand, the proposed process had clear roles and responsibilities. Thus, its execution was distributed between participants (among process phases), and its effort was minimized.

In addition to the empirical analysis, it is important to highlight that both the adopted TRL and the process contracts did not impose an additional effort to the proposed process. First, the proposed contracts were embedded into SORTT architecture. Thus, participants were not aware of each service's particularities and how they fulfilled the defined contracts. Also, TRL trace link representation was a major data structure which transited through process phases. Nevertheless, the proposed contracts and the adopted TRL facilitated the overall process workflow. They provided data structures for each process phase and established means of communication between them. Hence, the proposed process could be automated, improving its performance and mitigating the effort to trace artifacts.

Threats to Validity

As a final remark, it is important to mention the threats that we identified to validity of the experiment as well as how we addressed or mitigated them.

We minimized participants' history and maturation **internal threats** controlling the set of requirements per participant and also randomizing assigned requirements and processes. Our **conclusion threats** are minimized by the sample size, which counted on 129 test cases and 12 participants, which is a substantial number considering project's nature. Nevertheless, we highlight that experiment's participants were limited to the ones related to the evaluated project. Thus, the small number of participants and, consequently, experimental units may hinder the statistical power of our tests. Moreover, performance results indicate that the context of the experiment could not encompass complex scenarios, which could reflect critical tasks in software engineering and more complex traceability scenarios should be considered in future evaluations. Furthermore, project's particularities need to be considered, thus our obtained results cannot be externalized to other contexts.

Regarding external threats and experiment's limitations, we also highlight that our comparison did not considered industrial tools, such as IBM DOORS or Jazz. We did not compare industrial tools with SORTT due to different factors such as the time, procedures and effort to built an environment which could equally compare them. Regarding such factors,

we emphasize that most industrial tools demand one requirement traceability process centered on the tool, thus we could not distinguish the underlying traceability process from the tool itself. As a final note, we also highlight that experiment's validation did not consider a comparison with a requirements traceability process consolidated by maturity levels. Even though, obtained results are fruitful to the discussion of common aspects of the requirements traceability process.

6.3 Chapter Debriefings

In this chapter, we have presented the empirical evaluation of the proposed traceability language as well as the empirical evaluation of the proposed requirements traceability process.

To evaluate the proposed language, we have set up a controlled experiment, in which participants had to use different traceability query languages in order to retrieve different trace links. The experiment considered a variety of data sets, either from the literature or from industrial projects, and observed readability and writability criteria as well as how participants comprehended languages' trace link representation and queries. As a result, we observed that the TRL proposed trace link representation is prioritized in comparison with the other languages. Moreover, TRL queries were also prioritized and the language constructions were evaluated as easily read/written. Even though, the TracQL language was ranked as the most writable language and, despite not having the better trace link representation, TracQL's queries are also adequate to search and retrieve trace links. In such context, the small difference between the prioritization of TRL and TracQL queries did not provide statistical data to identify a language with a better outcome. Thus, both TracQL and TRL are feasible languages to provide an abstraction to requirements traceability.

To evaluate the proposed process, we have set up a controlled experiment, in which participants had to identify requirement related artifacts either following an ad hoc approach or following the proposed process. The experiment considered an industrial project and established trace links based on its test cases. As a result, we observed that the proposed process improves the performance and efficiency of the RT process, while maintaining the same accuracy of the ad hoc process. Notwithstanding, the experiment's setup and the analysis of its validity threats indicates that most of the performance gains are due to the process' tool

support. Therefore, the proposed process and its tool support are a feasible approach to improve requirements traceability. Moreover, the proposed process phases and contracts also foster the discussion of major aspects of the RT process, thus portable traceability can be addressed.

Chapter 7

Related Work

Requirements engineering and requirements traceability have been recognized as fundamental fields in the software engineering research area [40, 55]. In this context, Nuseibeh and Easterbrook had systematically reviewed the major contributions in the requirements engineering area [40], whereas Spanoudakis and Zisman had thoroughly revised the requirements traceability research field [55]. Considering the myriad of works in the requirements engineering and requirements traceability fields, in this chapter we discuss the fundamental works which are related to ours, either as a basis for our work or for its comparison.

In order to present the related work, first we list major works that address the challenges of requirements traceability (Section 7.1). Thus, providing a fundamental background related to requirements traceability and its challenges. Subsequently, we detail literature works which are related to requirements traceability languages (Section 7.2) as well as the ones related to requirements engineering process improvements (Section 7.3). Thus, concluding remarks are discussed (Section 7.4).

7.1 Traceability Challenges

In the late 90s, Gotel and Finkelstein were pioneers in the analysis of the requirements traceability problem [21]. Based on empirical studies, involving over 100 practitioners as well as an evaluation of presented tool support, they observed that most computer-aided software engineering (CASE) tools do not cover requirements traceability, and that the few ones that cover it suffer from problems of poor integration and inflexibility. Hence, environments

which integrate tools for all aspects of development enable requirements traceability throughout a project's life-cycle. In such context, a common language separating the representation of requirements would offer potential gains [21].

Succeeding Gotel and Finkelstein, Ramesh investigated the adoption of traceability practices in organizational environments [46]. By contrasting low-end and high-end requirements traceability users in an empirical experiment, he observed that in the absence of automated tools, traceability will not only be error prone and time consuming, but may be impossible to maintain [46]. Regarding the necessity of automated tools, he forecasts that process centered environments are a feasible approach to define the creation and the maintenance of traceable information [46].

In the beginning of the 21st, Kannenberg and Saiedian revisit the challenges involving requirements traceability [27]. Though much has changed, the size and complexity of new developed systems significantly influence requirements traceability and the cost of creating and maintaining traceable information. By observing organizational problems and inadequate tool support, it is discussed that the lack of well defined process and the view of requirements traceability as means of standards compliance hinder its correct utilization. In such context, commercial off-the-shelf (COST) tools provide only simplistic support for requirements traceability. Adopting COST requirements traceability tools require that one project's methodology must be centered around the tool workflow and, as previously observed by Gotel and Finkelstein, most of them are inflexible or lack integration with organizations' existing methodologies [27].

Eventually, in an attempt to bring light to the requirements traceability problems as well as classify research contributions and track progress in the field, the Center of Excellence of Software Traceability (CoEST) outlines eight challenges that needs to be addressed in order to achieve requirements traceability [22]. The eight challenges state that requirements traceability must be purposed, cost-effective, configurable, trusted, scalable, portable, valued and ubiquitous. Regarding such challenges and their research topics, as one of the basis of the present work, we consider the necessity to *(i)* design approaches to traceability based upon traceability abstractions, rather than concrete artifacts types, which can accommodate all the artifacts that are likely to arise in the life of a project *(ii)* standardize key aspects of the traceability process; and *(iii)* define traceability roles and responsibilities within a

traceability development contract, providing support for instantiating and discharging these in different project and organizational settings. Thus, we discuss requirements traceability in the scope of portable and scalable traceability.

7.2 Traceability Query Languages

Throughout literature, different languages have been used for traceability analysis in spite of their different aims. As examples, we cite SQL [47], OCL [10], and XML [35]. Additionally, domain specific languages have been proposed to directly address traceability needs, such as TracQL [57], TQL [34], or VTML [33]. Considering the vastness of languages used in the context of requirements traceability, and in order to compare them, we selected TracQL and TQL due to their completeness. Notice that, to the best of our knowledge, no previous work has explicitly compared traceability languages and we identified a single work that compared TracQL with domain specific languages (DSLs). Even though, the compared DSLs could be used in any domain problem rather than requirements traceability [57].

Regarding TracQL, a graph-based traceability query language, presented in Chapter 6, our approach mostly differ from theirs on: *(i)* the usage of an external DSL, explicitly declaring traced artifacts through TRL's constructions; and *(ii)* the definition of queries using expressions, instead of a graph like query structure. In TracQL, complex tasks require that TracQL queries traverse a graph accessing direct and indirect successors. Thus, writing complex queries in TracQL can be an elusive task. On the other hand, the TRL establishes direct links through requirements and artifacts, hence queries can be written independently of the existence of direct or indirect links. Regarding language extensibility, it is important to note that TracQL is the only evaluated language which provides it. TracQL provides extensibility since it is built on Scala, hence new types can be declared extending existing ones. On the other hand, TRL considers a language grammar and, as a consequence, extensibility is currently not possible without adding new constructions to the grammar. We also highlight that in order to measure readability and writability criteria, the evaluation of TracQL considered compiler tokens [57], whereas we considered human participants and a Likert scale. Additionally, we also considered an analytic hierarchy process comparison. Therefore, we believe that our work considers more factors that need to be acknowledged while comparing

traceability languages.

TQL, a traceability query language built on top of XML, and also introduced in Chapter 6, differs from our approach on: *(i)* the fact that abstractions are accessed through TQL functions, but are not explicitly declared, whereas TRL declares them; and *(ii)* the usage of source and target parameters in TQL queries. In such context, the separation of source and target locators can be elusive, since a trace link is a bidirectional relationship. On the other hand, our queries are based on the traceability definition proposed by Gotel and Finkelstein [21]. Therefore, trace links can be retrieved considering both forwards and backwards traceability. As a final remark, we highlight that in [34], TQL evaluation did not considered traceability benchmarks or an industrial project. The presented evaluation is limited to illustrating how TQL queries address a number of traceability questions. On the other hand, we infer our conclusions from an experiment, in which we have considered different data sets, with different types of trace links. Also, in order to better guarantee the soundness of our conclusions, our evaluation also considered possible threats to its validity.

As a final remark, and for the sake of completeness, we cite Mäder and Cleland-Huang proposal of a Visual Trace Modeling Language (VTML) [33]. In order to provide a high level abstraction to requirements traceability, their approach assumes the existence of and underlying meta-model, which is referred as the Traceability Information Model (TIM). Considering such model, VTML utilizes standard UML class diagrams to model traceability queries as a set of OCL constraints [11] enforced in the traceability meta-model. Such approach was evaluated comparing VTML queries with SQL ones. In such context, we were not able to empirically compare it with our approach due to setup hindrances. As VTML required a TIM, we were unable to faithfully construct such artifact in the context of the CoEST benchmarks or the e-Pol project, thus precluding that the language could be compared with the other ones. Nevertheless, we highlight that our evaluation considered similar criteria to theirs. More specifically, our language evaluation criteria, *i.e.* readability and writability, were based on theirs, since these are important factors measuring the usage of a traceability language.

Considering the discussion presented in this section, and the compared criteria, Table 7.1 summarizes the compared languages: TRL, TracQL, and TQL. TRL and TQL are external DSLs with specific grammars, whereas TracQL is an internal DSL built on Scala. Being an

Table 7.1: Related Work - Languages

Language	DSL	Extensibility	Evaluation	Readability/Writability	Portable	Scalable
TRL	external	no	empirical experiment	yes	partially	yes
TracQL	internal	yes	empirical experiment	yes	no	partially
TQL	external	no	illustrative examples	yes	no	no

internal DSL, TracQL is also the only language to provide extensibility, since new types can be declared extending existing ones. All languages were evaluated considering readability and writability criteria. TRL and TracQL were evaluated considering empirical experiments, whereas TQL presented illustrative examples to language's constructions. TRL partially addresses portable and scalable traceability. Portable traceability is partially addressed since a common trace link representation must be adopted, whereas scalable traceability is partially addressed through the design and implementation of new services. On the other hand, TracQL partially addresses scalable traceability, since new types and functions can extend the language. Finally, TQL do not consider portable or scalable traceability.

7.3 Requirements Engineering Process Improvements

Requirements traceability was recognized as a key process area [51], even though few works have studied its particularities in the context of requirements engineering process improvements. To the best of our knowledge, the majority of the works on such field focus on *(i)* the aspects of elucidating system requirements [24, 40, 59]; and *(ii)* how to understand the dynamics and particularities of such process [20, 53, 61]. Nevertheless, we discuss major contributions to process improvements considering the requirements traceability research scope.

Sawyer *et al.* thoroughly presents and details software process improvement practices in the scope of requirements engineering [51]. Notwithstanding, they observed that the requirements process is much less homogeneous and well understood than the software development process as a whole. Additionally, the necessity to identify common aspects of the requirements traceability process is a major research field, related to the grand challenges of requirements traceability [22]. Therefore, these researches are one of the basis of our work. We consider these issues and propose a requirement traceability process centered on

the traceability model, thoroughly discussing process phases, activities, actors, responsibilities and input/output artifacts. Notwithstanding, we do not provide a definitive solution to the heterogeneity of traceability processes or to portable traceability. Even though, the investigation presented on this dissertation can foster future works which address this research field, *e.g.* the definition of contracts which govern the requirements traceability process phases.

Considering traceability grand challenges, Gotel *et al.* proposed a generic traceability process [22]. Throughout their work, process key phases and some of its activities are briefly described. However, they neither detailed such process nor specified how the process phases/activities would interoperate. In such context, our work differs from theirs on: *(i)* the usage of a common trace link representation through process phases; *(ii)* the delineation of process phases as well as its major activities; *(iii)* the definition of process' actors, responsibilities, and inputs/outputs; and *(iv)* the definition of contracts, which established how the process phases/activities would exchange data. As a final remark, we highlight that their process was discussed without a concrete case study or evaluation, whereas, in order to support our conclusions, we presented an empirical investigation of our process, in which we have considered an industrial project as the corpus of the study. Additionally, we presented SORTT, a service oriented requirements traceability tool, which supports the proposed process and automates part of its activities.

Ramesh and Jarke proposed four reference models in order to provide a modeling framework for requirements traceability [45]. Their models specified requirements management and requirements rationale as well as declared the relationships between requirements and system components/design. Although, the necessary activities to implement the models were not detailed and their work focused on how the models were conceived. In such context, our work is related to their requirements and components/design reference model, however we mostly differ from theirs on the scrutinization of or our proposed process and its activities. Notice that, even when our proposed process can be used in different contexts, such approach cannot address all organizations and traceability needs, which reinforces the conception of different reference models, as discussed by Ramesh and Jarke. Finally, we highlight that their approach was automated by means of SLATE, a System Level Automation Tool for Engineers, whereas our approach is backed by SORTT. SLATE differs form SORTT due to the fact that *(i)* it is more focused in high level requirements tracing, such as requirements

Table 7.2: Related Work - Processes

Language	Defined trace link representation	Detailed phases and activities	Roles and responsibilities	Traceability contracts	Tool Support
TRL	yes	yes	yes	yes	yes
Gotel et al.	no	partially	partially	partially	no
Ramesh and Jarke	no	partially	yes	no	yes

management and requirements rationale, and *(ii)* all traced artifacts and requirements are declared directly in the tool, and need to be maintained through its functionalities. On the other hand, SORTT *(i)* focuses on requirements to system artifacts, and *(ii)* extracts traces directly from produced artifacts, which are maintained independently of the tool and are abstracted through TRL's constructions.

Considering the discussion presented in this section, and the compared criteria, Table 7.2 summarizes the compared processes: TRL based process, Ramesh and Jarke [45], and Gotel *et al.* [22]. Regarding the compared criteria, it is important to emphasize that the TRL based process considers traceability contracts which provide interfaces to its activities. In such context, Gotel *et al.* also emphasize the necessity of traceability contracts, but do not propose or detail them. On the other hand, Ramesh and Jarke do not consider such interfaces.

7.4 Chapter Debriefings

In this chapter, we have presented a structured review of requirements traceability works in the scope of requirements traceability challenges, languages, and process improvements. In such context, we have observed problems related to the heterogeneity of traceability processes as well as the lack of an integrated tool support, which could accommodate organizations' specific needs. Therefore, we have considered the necessity to provide abstractions to requirements traceability, standardize key aspects of the requirements traceability process, and define requirements traceability process, thus proposing a traceability representation language and a traceability process, centered on the traceability model. Moreover, considering factors such as the cost and time related to manual traceability as well as the lack of adequate tool support, we designed SORTT, a service oriented requirements traceability tool, which integrates different customizable services related to different phases of the require-

ments traceability process.

Chapter 8

Conclusions

In this work, we considered the necessity to provide abstractions to requirements traceability and also the necessity to standardize key aspects of the requirements traceability process considering the traceability model and also tracing requirements to system artifacts. Therefore, we presented an approach to represent traceable information through a traceability representation language (TRL). Such language is then exploited through a traceability process, traversing its phases, and being a major artifact in requirements traceability contracts.

The presented approach considers Gotel and Finkelstein requirements traceability definition and presents a declarative language through which requirements and artifacts are represented. Moreover, the proposed language also provides means to specify queries, which search and retrieve the relationships between declared requirements and artifacts. By means of the proposed language, and also by considering the traceability model, we presented a traceabily process centered on them. Thus, we detailed process' phases, activities, actor, and responsibilities as well as presented requirements traceability contracts, which govern process' phases and how they exchange information. Finally, as a tool support, we presented SORTT, a Service Oriented Requirements Traceability Tool, which automates part of the activities detailed in the proposed process.

In order to evaluate the proposed approach, we elaborated two empirical studies, considering benchmarks from the Center of Excellence for Software Traceability and also an industrial project being developed by the Federal University of Campina Grande, analyzing both the proposed language and the proposed process.

Regarding the proposed traceability representation language (TRL), we investigated fac-

tors related to its readability and writability. To this extent, it was compared to traceability query languages, namely TracQL and TQL. As a result, we observed that TRL is prioritized in comparison with the other languages. TRL queries were prioritized and the language constructions were evaluated as easily read/written. Though, TracQL also presents as an adequate language, in which TracQL as evaluated as easily read/written. Therefore, TRL and TracQL are feasible approaches to provide an abstraction to requirements traceability.

Additionally, the proposed requirements traceability process was compared to an ad hoc process in the context of the industrial project. To this extent, factors such as the performance and effectiveness of the compared process were analyzed and, as a result, we observed that in comparison with the ad hoc process, the proposed process has a better performance and efficiency due to its provided tool support. On the other hand, both the ad hoc process and the proposed process had the same accuracy. Even though there were no highly significant quantitative gains through the usage of the proposed process, qualitative analysis was fruitful. The discussion of a traceability process considering the defined phases present in the traceability model as well as the design of requirements traceability contracts foster the discussion of major aspects of the requirements traceability process, thus portable traceability can be addressed.

8.1 Contributions

In short, the present work's contributions are summarized as follows:

- Proposition of an approach to abstract traceable information through a traceability representation language (TRL);
- Evaluation of the proposed language in different requirement traceability projects, with heterogeneous types of traceable information;
- Proposition of a requirements traceability process centered on the traceability model;
- Definition of requirements traceability contracts which establish means of communication through requirements traceability process' phases;
- Design of a service oriented requirements traceability tool, namely SORTT;

- Evaluation of the proposed process in an industrial project under development.

8.2 Limitations

As a final remark, it is important to emphasize the limitations of the presented work.

Cosidering the proposed TRL, it is important to highlight that the language does not provide extensibility, and new constructions demand the extension of the language's grammar. Moreover, the languages focuses on providing abstractions to requirements and system artifacts, although it currently supports only one level of granularity, *i.e.* artifacts or requirements cannot be grouped such that the language provides different levels of granularity, from coarse-grained to fine-grained traceability [40, 43]. As a final note, since the language focuses on providing abstractions to requirements and system artifacts, its query mechanism is centered on traceability in forward and backward directions, which may not be adequate to some tasks supported by requirements traceability.

Regarding the proposed process and, more specifically, the production of trace links, we highlight that the present work do not propose a new technique to produce them. In fact, the production of trace links depends on already existing techniques and, as a consequence, it is suitable to the same faults and problems that they have [3, 16, 21, 27, 46]. Such techniques focus on the automatic production of trace links thus, instead of considering the whole traceability process, they exclusively encompass activities of the production phase. In such context, they could be adapted and exploited by the proposed process. For instance, Egyed and Grünbacher [18] propose the production of trace links based on test scenarios. Considering a minimal set of trace links extracted from these scenarios, new ones are inferred based on trace analysis techniques. Hence, their approach could be inserted in the production phase of the proposed process, *i.e.* creating the test scenarios, tagging them with tested requirements, and also producing and inferring trace links. Moreover, factors related to the maintainability of trace links are not addressed. Even that our proposed approach extracts traces links directly from artifacts, if the artifacts are not correctly updated, incorrect trace links can be extracted [27, 46].

8.3 Future Work

Considering presented contributions, a wide variety of research questions and topics remains to be investigated. Most of these topics are related to the ones discussed throughout the grand challenges of requirements traceability [22]. Notwithstanding, in this section, we highlight interesting research fields related to the present work.

First, it is possible to perform a more complete evaluation of traceability languages and their expressiveness. For such evaluation, one could implant the evaluated languages in different organizations, observing how they suit organizations' needs and also how the languages are used through the life-cycle of a project.

Another interesting research could consider a more complete evaluation of traceability processes. For instance, one could consider factors such as the scalability of traceability processes and how different artifact types can be plugged to them. Moreover, one could observe how traceability processes comply with standards or regulatory measures, defining metrics and means to compare traceability processes.

Regarding traceability processes, an in depth evaluation of traceability contracts is an interesting research topic. Considering that traceability contracts provide interfaces between process activities, access the benefits and the gains that the contracts could provide to traceability processes was superficially addressed in this work. Therefore, further investigating the traceability contracts is a promising research topic. Moreover, access how trace links maintenance can be addressed is a second open research topic. The discussion in the present work briefly considered trace links maintenance, though this is a depth research field, with challenges and particularities that requires further research.

Finally, an empirical evaluation of industrial tools and their underlying traceability processes is indeed a fruitful investigation field. By evaluating traceability tools as well as practitioners' thoughts [41], one can identify the actual needs of requirements traceability and envision a new generation of traceability tools.

8.4 Final Remarks

Considering the grand challenges of requirements traceability, and more specifically, the research topics investigated in the present work, our study presents initial steps towards portable and scalable traceability. Notwithstanding, a wide variety of research questions and topics remains to be investigated. Hence, the present work is an attempt to enlighten possible approaches which address part of the requirement traceability problems.

Bibliography

[1] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo. Recovering traceability links between code and documentation. *Software Engineering, IEEE Transactions on*, 28(10):970–983, 2002.

[2] P. Arkley and S. Riddle. Overcoming the traceability benefit problem. In *Requirements Engineering, 2005. Proceedings. 13th IEEE International Conference on*, pages 385–389, Aug 2005.

[3] Paul Arkley, Paul Mason, and Steve Riddle. Position Paper: Enabling Traceability. *Proceedings of 1st TEFSE*, 2002.

[4] N. Assawamekin, T. Sunetnanta, and C. Pluempitiwiriyawej. Resolving multiperspective requirements traceability through ontology integration. In *Semantic Computing, 2008 IEEE International Conference on*, pages 362–369, 2008.

[5] N. Assawamekin, T. Sunetnanta, and C. Pluempitiwiriyawej. Deriving traceability relationships of multiperspective software artifacts from ontology matching. In *Software Engineering, Artificial Intelligences, Networking and Parallel/Distributed Computing, 2009. SNPD '09. 10th ACIS International Conference on*, pages 549–554, 2009.

[6] N. Assawamekin, T. Sunetnanta, and C. Pluempitiwiriyawej. Mupret: An ontology-driven traceability tool for multiperspective requirements artifacts. In *Computer and Information Science, 2009. ICIS 2009. Eighth IEEE/ACIS International Conference on*, pages 943–948, 2009.

[7] Victor R. Basili, Gianluigi Caldiera, and H. Dieter Rombach. The goal question metric approach. In *Encyclopedia of Software Engineering*. Wiley, 1994.

[8] H. D. Benington. Production of large computer programs. In *Proceedings of the 9th International Conference on Software Engineering*, ICSE 87, pages 299–310, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press.

[9] Keith H. Bennett and Vaclav T. Rajlich. Software maintenance and evolution: A roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, pages 73–87, New York, NY, USA, 2000. ACM.

[10] Lionel C. Briand, Yvan Labiche, and Tao Yue. Automated traceability analysis for {UML} model refinements. *Information and Software Technology*, 51(2):512 – 527, 2009.

[11] Jordi Cabot and Martin Gogolla. Object constraint language (ocl): A definitive guide. In *Proceedings of the 12th International Conference on Formal Methods for the Design of Computer, Communication, and Software Systems: Formal Methods for Model-driven Engineering*, SFM'12, pages 58–90, Berlin, Heidelberg, 2012. Springer-Verlag.

[12] Mary Beth Chrissis, Mike Konrad, and Sandy Shrum. *CMMI Guidlines for Process Integration and Product Improvement*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[13] J. Cleland-Huang, C.K. Chang, and M. Christensen. Event-based traceability for managing evolutionary change. *Software Engineering, IEEE Transactions on*, 29(9):796–810, 2003.

[14] J. Cleland-Huang, C.K. Chang, G. Sethi, K. Javvaji, Haijian Hu, and Jinchun Xia. Automating speculative queries through event-based requirements traceability. In *Proceedings of the 10th IEEE RE*, pages 289–296, 2002.

[15] A De Lucia, F. Fasano, R. Oliveto, and G. Tortora. Enhancing an artefact management system with traceability recovery features. In *Proceedings of the 20th IEEE ICSM*, 2004.

[16] Ralf Dömges and Klaus Pohl. Adapting traceability environments to project-specific needs. *Commun. ACM*, 41(12):54–62, December 1998.

[17] Alexander Egyed and Paul Grunbacher. Supporting software understanding with automated requirements traceability. *International Journal of Software Engineering and Knowledge Engineering*, 15(05):783–810, 2005.

[18] Alexander Egyed and Paul GRünbacher. Supporting software understanding with automated requirements traceability. *International Journal of Software Engineering and Knowledge Engineering*, 15(05):783–810, 2005.

[19] Daniel Gondim. Uma abordagem para construcao das etapas de analise de um compilador. Master's thesis, Universidade Federal de Campina Grande, Campina Grande, 2014.

[20] Tony Gorschek and Alan M. Davis. Requirements engineering: In search of the dependent variables. *Information and Software Technology*, 50(1–2):67 – 75, 2008.

[21] O. C Z Gotel and A. C W Finkelstein. An analysis of the requirements traceability problem. In *Proceedings of the 1st IEEE RE*, 1994.

[22] Orlena Gotel, Jane Cleland-Huang, JaneHuffman Hayes, Andrea Zisman, Alexander Egyed, Paul Grunbacher, Alex Dekhtyar, Giuliano Antoniol, and Jonathan Maletic. The grand challenge of traceability. In *Software and Systems Traceability*. Springer, 2012.

[23] Dick Grune. *Parsing Techniques: A Practical Guide*. Springer Publishing Company, Incorporated, 2nd edition, 2010.

[24] Noriko Hanakawa and Masaki Obana. A metrics for meeting quality on a software requirement acquisition phase. In Oscar Dieste, Andreas Jedlitschka, and Natalia Juristo, editors, *Product-Focused Software Process Improvement*, volume 7343 of *Lecture Notes in Computer Science*, pages 260–274. Springer Berlin Heidelberg, 2012.

[25] J.H. Hayes, A. Dekhtyar, and J. Osborne. Improving requirements tracing via information retrieval. In *Proceedings of the 11th IEEE RE*, pages 138–147, 2003.

[26] E. Iee. Ieee recommended practice for software requirements specifications. Technical report, IEEE, 1998.

[27] A. Kannenberg and H. Saiedian. Why software requirements traceability remains a challenge. *The Journal of Defense Software Engineering*, 2009.

[28] AndrewJ. Ko, ThomasD. LaToza, and MargaretM. Burnett. A practical guide to controlled experiments of software engineering tools with human participants. *Empirical Software Engineering*, 2013.

[29] Tomaz Kosar, Marjan Mernik, and JeffreyC. Carver. Program comprehension of domain-specific and general-purpose languages: comparison using a family of experiments. *Empirical Software Engineering*, 2012.

[30] C. Larman and V.R. Basili. Iterative and incremental developments. a brief history. *Computer*, 36(6):47–56, June 2003.

[31] James Law and Gregg Rothermel. Incremental dynamic impact analysis for evolving software systems. In *Proceedings of the 14th International Symposium on Software Reliability Engineering*, ISSRE '03, pages 430–, Washington, DC, USA, 2003. IEEE Computer Society.

[32] B. P. Lientz, E. B. Swanson, and G. E. Tompkins. Characteristics of application software maintenance. *Commun. ACM*, 21(6):466–471, June 1978.

[33] Patrick Mader and Jane Cleland-Huang. A visual traceability modeling language. In *Proceedings of the 13th International Conference on Model Driven Engineering Languages and Systems: Part I*, MODELS 10, pages 226–240, Berlin, Heidelberg, 2010. Springer-Verlag.

[34] J.I Maletic and M.L. Collard. Tql: A query language to support traceability. In *5th ICSE Workshop on TEFSE*, 2009.

[35] Jonathan I. Maletic, Michael L. Collard, and Bonita Simoes. An xml based approach to support the evolution of model-to-model traceability links. In *Proceedings of the 3rd international workshop on Traceability in emerging forms of software engineering*, TEFSE '05, pages 67–72, New York, NY, USA, 2005. ACM.

[36] Arthur Marques, Franklin Ramalho, and Wilkerson L. Andrade. An approach to represent trace links considering portable and scalable traceability. Submitted to the Brazilian Symposium on Software Engineering (SBES), April 2014.

[37] Arthur Marques, Franklin Ramalho, and Wilkerson L. Andrade. Towards a requirements traceability process centered on the traceability model. Submitted to The 15th International Conference of Product Focused Software Development and Process Improvement (PROFES), June 2014.

[38] Arthur Marques, Franklin Ramalho, and Wilkerson L. Andrade. Towards a requirements traceability process centered on the traceability model. Submitted to the ACM Symposium on Applied Computing (SAC), September 2014.

[39] Arthur Marques, Franklin Ramalho, and Wilkerson L. Andrade. TRL – a traceability representation language. Submitted to the ACM Symposium on Applied Computing (SAC), September 2014.

[40] Bashar Nuseibeh and Steve Easterbrook. Requirements engineering: A roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, pages 35–46, New York, NY, USA, 2000. ACM.

[41] Marian Petre. Uml in practice. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 722–731, Piscataway, NJ, USA, 2013. IEEE Press.

[42] John L. Pfaltz. Using concept lattices to uncover causal dependencies in software. In Rokia Missaoui and Jurg Schmidt, editors, *Formal Concept Analysis*, volume 3874 of *Lecture Notes in Computer Science*, pages 233–247. Springer Berlin Heidelberg, 2006.

[43] Francisco A.C. Pinheiro. Requirements traceability. In Julio Cesar Sampaio Prado Leite and Jorge Horacio Doorn, editors, *Perspectives on Software Requirements*, volume 753 of *The Springer International Series in Engineering and Computer Science*, pages 91–113. Springer US, 2004.

[44] Klaus Pohl. *Requirements Engineering: Fundamentals, Principles, and Techniques*. Springer Publishing Company, Incorporated, 1st edition, 2010.

[45] B. Ramesh and M. Jarke. Toward reference models for requirements traceability. *Software Engineering, IEEE Transactions on*, 27(1):58–93, 2001.

[46] Balasubramaniam Ramesh. Factors influencing requirements traceability practice. *Commun. ACM*, 1998.

[47] S.P. Reiss. Incremental maintenance of software artifacts. *Software Engineering, IEEE Transactions on*, 2006.

[48] C. J. Van Rijsbergen. *Information Retrieval*. Butterworth-Heinemann, Newton, MA, USA, 2nd edition, 1979.

[49] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual, The (2Nd Edition)*. Pearson Higher Education, 2004.

[50] T.L. Saaty. *Fundamentals of the Analytic Hierarchy Process*. RWS Publications, 2000.

[51] Pete Sawyer, Ian Sommerville, and Stephen Viller. Requirements process improvement through the phased introduction of good practice. In *Software Process - Improvement and Practice*, pages 31–44, 1997.

[52] Ian Sommerville. *Software Engineering: (Update) (8th Edition) (International Computer Science)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[53] Ian Sommerville and Jane Ransom. An empirical study of industrial requirements engineering process assessment and improvement. *ACM Trans. Softw. Eng. Methodol.*, 14(1):85–117, January 2005.

[54] George Spanoudakis. Plausible and adaptive requirement traceability structures. In *Proceedings of the 14th international conference on Software engineering and knowledge engineering*, SEKE '02, pages 135–142, New York, NY, USA, 2002. ACM.

[55] George Spanoudakis and Andrea Zisman. Software traceability: A roadmap. In *Handbook of Software Engineering and Knowledge Engineering*. World Scientific Publishing, 2004.

[56] George Spanoudakis, Andrea Zisman, Elena Pérez-Minana, and Paul Krause. Rule-based generation of requirements traceability relations. *Journal of Systems and Software*, 72(2):105 – 127, 2004.

[57] N. Tausch, M. Philippsen, and J. Adersberger. Tracql: A domain-specific language for traceability analysis. In *10th Working IEEE WICSA Conference*, 2012.

[58] Richard Torkar, Tony Gorschek, Robert Feldt, Mikael Svahnberg, Uzair Akbar Raja, and Kashif Kamran. Requirements traceability: a systematic review and industry case study. *International Journal of Software Engineering and Knowledge Engineering*, 22(3):385–434, 2012.

[59] Toshihiko Tsumaki and Tetsuo Tamai. Framework for matching requirements elicitation techniques to project characteristics. *Software Process: Improvement and Practice*, 11(5):505–519, 2006.

[60] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Bjöorn Regnell, and Anders Wesslén. *Experimentation in software engineering: an introduction*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[61] A.S.Aminah Zawedde, M.D.Martijn Klabbers, D.Ddembe Williams, and M.G.J.Mark van den Brand. Understanding the dynamics of requirements process improvement: A new approach. In Danilo Caivano, Markku Oivo, MariaTeresa Baldassarre, and Giuseppe Visaggio, editors, *Product-Focused Software Process Improvement*, volume 6759 of *Lecture Notes in Computer Science*, pages 276–290. Springer Berlin Heidelberg, 2011.

[62] C. Ziftci and I. Krueger. Tracing requirements to tests with high precision and recall. In *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*, pages 472–475, Nov 2011.

[63] Celal Ziftci. *Mining Test Cases To Improve Software Maintenance*. PhD thesis, University of California, San Diego, 2013.

Appendix A

Language's Evaluation Questionnaire

For the sake of simplicity, and also due to the author's **environmental awareness**, the questionnaire is available online[1]. It's important to emphasize that, even that the questionnaire is accepting new responses, this is a sibling questionnaire of the original one.

Analogously, questionnaire's summary of responses[2] and also results analyses[3] are available online.

[1] http://goo.gl/JxxTfM
[2] http://goo.gl/Asi3AB
[3] http://goo.gl/ylFO90

Appendix B

Process Evaluation Results

Table B.1: Process Evaluation - Overall Results

Participant	Process	Requirement	Time	Precision	Recall	F-measure	Efficiency
P_1	AD	UC12	21	1	1	1	0.04
	PP	UC05	8.63	1	1	1	0.11
P_2	AD	UC05	8	1	0.66	0.8	0.1
	PP	UC12	4.95	1	1	1	0.20
P_3	AD	UC03	15.5	1	1	1	0.06
	PP	UC24	7.57	1	1	1	0.13
P_4	AD	UC25	18	1	1	1	0.05
	PP	UC05	4.95	1	1	1	0.20
P_5	AD	UC24	4.49	1	0.5	0.66	0.14
	PP	UC03	6.53	1	1	1	0.15
P_6	AD	UC05	17.49	0.95	1	0.97	0.055
	PP	UC25	4.75	1	1	1	0.21
P_7	AD	UC01	21.4	1	1	1	0.04
	PP	UC03	7.75	1	1	1	0.12
P_8	AD	UC25	15.5	0.63	1	0.77	0.05
	PP	UC01	7.63	1	1	1	0.13
P_9	AD	UC12	11	1	1	1	0.09
	PP	UC01	6.92	1	1	1	0.14
P_{10}	AD	UC01	12.29	1	1	1	0.08
	PP	UC12	5.91	1	1	1	0.16
P_{11}	AD	UC24	14.3	1	1	1	0.06
	PP	UC13	7.28	1	1	1	0.13
P_{12}	AD	UC13	12.2	1	1	1	0.08
	PP	UC24	4.65	1	1	1	0.21