INAUGURAL – DISSERTATION

submitted

to the

Combined Faculty for the Natural Sciences and Mathematics

of

Heidelberg University, Germany

for the degree of

Doctor of Natural Sciences

Put forward by

M.Sc. Paul Hübner

Born in Kulmbach

Oral examination: .....

Interaction-Based Creation and Maintenance of Continuously Usable Trace Links

Supervisor: Prof. Dr. Barbara Paech (Heidelberg University)
Advisor: Prof. Dr. Patrick Mäder (JP) (Technical University of Ilmenau)

Abstract

Traceability is a major concern for all *Software Engineering* (*SE*) artefacts. The core of traceability are trace links between the artefacts. Out of the links between all kinds of artefacts, trace links between requirements and source code are fundamental, since they enable the connection between the user point of view of a requirement and its actual implementation. Trace links are important for many *SE* tasks such as maintenance, program comprehension, verification, etc. Furthermore, the direct availability of trace links during a project improves the performance of developers.

The manual creation of trace links is too time-consuming to be practical. Thus, traceability research has a strong focus on automatic trace link creation. The most common automatic trace link creation methods use *Information Retrieval* (*IR*) techniques to measure the textual similarity between artefacts. The results of the textual similarity measurement is then used to judge the creation of links between artefacts. The application of such *IR* techniques results in a lot of wrong link candidates and requires further expert knowledge to make the automatically created links usable, insomuch as it is necessary to manually vet the link candidates. This fact prevents the usage of *IR* techniques to create trace links continuously and directly provide them to developers during a project.

Thus, this thesis addresses the problem of continuously providing trace links of a good quality to developers during a project and to maintain these links along with changing artefacts. To achieve this, a novel automatic trace link creation approach called *Interaction Log Recording-based Trace Link Creation* (*ILog*) has been designed and evaluated. *ILog* utilizes the interactions of developers with source code while implementing requirements. In addition, *ILog* uses the common development convention to provide issues' identifiers in a commit message, to assign recorded interactions to requirements. Thus *ILog* avoids additional manual efforts from the developers for link creation.

ILog has been implemented in a set of tools. The tools enable the recording of interactions in different *Integrated Development Environments* (*IDEs*) and the subsequent creation of trace links. Trace link are created between source code files which have been touched by interactions and the current requirement which is being worked on. The trace links which are initially created in this way are further improved by utilizing interaction data such as interaction duration, frequency, type, etc. and *Source Code Structure* (*SCS*), i.e. source code references between source code files involved in trace links. *ILog's* link improvement removes potentially wrong links and subsequently adds further correct links.

ILog was evaluated in three empirical studies using gold standards created by experts. One of the studies used data from an open source project. In the two other studies, student projects involving a real world customer were used. The results of the studies showed that *ILog* can create trace links with perfect precision and good

recall, which enables the direct usage of the links. The studies also showed that the *ILog* approach has better precision and recall than other automatic trace link creation approaches, such as *IR*.

To identify *Trace Link Maintenance* (*TM*) capabilities suitable for the integration in *ILog*, a *Systematic Literature Review* (*SLR*) about *TM* was performed. In the *SLR* the *TM* approaches which were found are discussed on the basis of a standardized *TM* process. Furthermore, the extension of *ILog* with suitable *TM* capabilities from the approaches found is illustrated.

Zusammenfassung

Rückverfolgbarkeit ist für alle Artefakte im *Software Engineering* (*SE*) wichtig, Verbindungen zwischen Artefakten bilden hierfür die Grundlage. Besonders wichtig sind Verbindungen zwischen Anforderungen und Quellcode, da diese die Verbindung zwischen der Nutzersicht in den Anforderungen und deren Implementierung darstellen. Diese Verbindungen sind für viele Aufgaben im *SE*, wie z.B. Wartung, Programmverständnis, Verifikation, etc. wichtig. Stehen solche Verbindungen direkt während der Softwareentwicklung zur Verfügung, wirkt sich dies positiv auf die Produktivität aus, da Aufgaben effizienter bearbeitet werden können.

Manuelles Erstellen von Artefakt-Verbindungen ist jedoch u.a. wegen des dafür notwendigen Zeitaufwands nicht praktikabel. Daher widmet sich die Rückverfolgbarkeits-Forschung intensiv Methoden zur automatischen Verbindungserstellung. Hierbei verwenden die meisten Ansätze *Information Retrieval* (*IR*) Techniken, um durch textuelle Ähnlichkeit von Artefakten zusammengehörige Artefakte zu ermitteln und zu verbinden. Das Verwenden von *IR* Techniken zur Verbindungserstellung führt zum Erstellen vieler falscher Verbindungen und erfordert deren nachträgliche manuelle Überprüfung durch Experten, um die Verbindungen nutzen zu können. Dies führt dazu, dass es nicht möglich ist *IR* Techniken zur fortlaufenden Verbindungserstellung und zur direkten Nutzung der erstellten Verbindungen einzusetzen.

Die vorliegende Arbeit erforscht daher das fortlaufende Erstellen von Verbindungen zwischen Artefakten, mit dem Ziel Verbindungen mit so guter Qualität zu erstellen, dass diese direkt genutzt werden können. Außerdem wird das mit sich ändernden Artefakten einhergehende Warten der Verbindungen erforscht. Dafür wurde die neue automatische *Interaktionsdaten-aufzeichnungsbasierte-Verbindungserstellung (ILog)* entwickelt und evaluiert. *ILog* nutzt die Interaktionen von Entwicklern mit Quellcodedateien während diese Anforderungen implementieren. Zusätzlich nutzt *ILog* die Konvention, dass in Übertragungsnachrichten von Versionskontrollsystemen Aufgabenkennungen angegeben werden. Mittels der Aufgabenkennungen können aufgezeichnete Interaktionen direkt Anforderungen zugeordnet werden, dies vermeidet zusätzlichen manuellen Aufwand für die Verbindungserstellung.

ILog wurde in mehreren Werkzeuge implementiert, diese ermöglichen das Aufzeichnen von Interaktionen in unterschiedlichen *Entwicklungsumgebungen* und die anschließende Erstellung von Verbindungen. Hierbei werden Verbindungen zwischen Dateien, mit denen interagiert wurde, und der Anforderung, an welcher währenddessen gearbeitet wurde, erstellt. Die initial so erstellten Verbindungen werden anschließend optimiert. Für die Optimierung werden sowohl interaktionsspezifische Daten, d.h. Dauer, Häufigkeit, Typ etc., als auch die Quellcodestruktur, d.h. Quellcodereferenzen zwischen Quellcodedateien, die über Artefakt-Verbindungen bereits mit Anforderungen verbunden sind, verwendet. Die Optimierung entfernt potentiell falsche Verbindungen und erstellt weitere bisher nicht gefundene Verbindungen.

ILog wurde in drei empirischen Studien mit von Experten erstellten Goldstandards evaluiert. In einer der Studien wurden Daten eines Open Source Projektes genutzt. In den beiden anderen Studien wurden Projekte mit Studierenden als Entwicklern und einem externen Kunden verwendet. Die Ergebnisse der Studien zeigen, dass *ILog* Verbindungen mit sehr hoher Genauigkeit und hoher Trefferquote erzeugen kann und so das direkte Nutzen der Verbindungen ermöglicht. Die Studien zeigen auch, dass *ILog* eine höhere Genauigkeit und Trefferquote als andere Ansätze zur Verbindungserstellung bietet (bspw. Ansätze, die *IR* Techniken verwenden).

Zur Integration der Wartung von Verbindungen in *ILog* wurde zunächst eine *systematische Literaturrecherche* zu Ansätzen für die *Wartung von Verbindungen* durchgeführt. In den Ergebnissen der Recherche werden gefundene Ansätze zur *Wartung von Verbindungen* mit einem dafür entwickelten standardisierten Prozess beschrieben und diskutiert. Darüber hinaus wird die Erweiterung von *ILog* mit Fähigkeiten zur *Wartung von Verbindungen* aus den gefundenen Ansätzen aufgezeigt.

Acknowledgements

For seven years I've proudly worked as a researcher with ups and downs to finally come to the point to write these acknowledgements!

First and foremost, I would like to thank Prof. Dr. Barbara Paech for her supervision over the last years. She always supported me during my thesis, was eager to discuss and give valuable feedback, and, in a way which in retrospect still surprises me, no matter what, was always patient with me! I am very grateful for the opportunity to work and be a software engineering researcher in her Software Engineering Group at the Faculty of Mathematics and Computer Science of Heidelberg University. Furthermore, I would like to thank my advisor Prof. Dr. Patrick Mäder for his helpful suggestions and his time for evaluating my thesis. I still remember back in 2015 when I met with Patrick in person for the first time at a conference, and the first thing he said to me was to ask him any questions I had about traceability research at any time! During the last few years, we met several times at different conferences and Patrick was always a great discussant when present at my talks.

Of course many more people have been involved in the working process around this thesis. To begin with, these are my fellows at the software engineering research group. In no particular, but in some way a chronological, order I'd like to mention Alexander, who introduced me into the world of university teaching. Tom, who was my first and all-time best office mate, the most clever and polite guy I've ever met. Gabi, who performed my first software engineering field studies and exchanged students with me, while being laid-back in any situation. Rumi, who passed her office table including two screens to me and randomly appeared all of a sudden for long-lasting fundamental and profound discussions. Thorsten, my loyal co-author, who introduced me to how to really do empirical research, including its drawbacks and complete changes in direction. Thomas, who showed me how to write research papers in a good way and kinda helped me to find my own research path. Marcus, who always backed me up as well, and always, in an ad hoc manner whenever it was necessary, to support me to get the job done. Christian, who never stopped discussing and answered my questions before I could express them. Anja, who showed me how to use software and inspired my research. Astrid, who took over students' teaching with amazing speed. Willi, who took me as his Padawan student during all these years, and Doris, who organized my researching and teaching life and enriched my being far beyond that. It's been such a pleasure to be the colleagues of one and all of you!

Moreover, I'd also like to mention the students with whom I worked during my time at Heidelberg University. Arthur, Thorsten, Carsten, Philipp and Viktor, you have all at least taught me how to program, and all of you, as well as those of you not mentioned by name here, have contributed to this thesis.

There were several further people who have inspired me for doing software engineering research, some of whom I want to mention now. First, Dan Berry, who was present at all my conference talks (and paper reviews) and who gave me important insights as to how to defend my point of view, starting from my very first doctoral symposium speech. Long before that, during my undergraduate student life, there was Hans-Jörg Happel, for whom I worked as a student assistant and who supervised my first software engineering research thesis. With his fascinating research attitude to come up with new ideas and to apply them in research, he was and still is my ideal inspiration to become a software engineering researcher. Furthermore, there have been several fellow students and friends of mine who encouraged me to get into and during my computer science studies. However, the list is too long to mention everyone by name, but you'll know who I mean anyway!

And lastly, I'd like to mention my family and close relatives! You have all supported me during and before the years of this thesis, no questions asked, whatever unconventional directions I'd choose, with at least all the support I needed to follow my path. It was the affinity for innovation and technology of my Dad who brought me to my first computer almost 30 years ago, and thus finally led me into taking on this thesis. My Mum inspired me with her attitude to never hesitate even if the upcoming is a big unknown. My siblings, Eva, Christoph and Thomas, my sister-in-law Damaris and my nephew Amos, were all never too tired to discuss my research during the past years with me, and somehow it even seems they understood what it is all about. Thank you for your support and attention! That's all for now from my side, don't forget to read the thesis and ask me questions.

Bye for now!

Contents

List of Figures

List of Tables

List of Abbreviations

API	Application Programming Interface
ComL	Version Control System *(*VCS*)* Commit-based Trace Link Creation
FN	False Negative
FP	False Positive
GS	Gold Standard
IDE	Integrated Development Environment
IL	ILog with manual performed interaction assignment
IL_{Com}	ILog with VCS Commit-based interaction assignment
ILog	Interaction Log Recording-based Trace Link Creation
IR	Information Retrieval
ITS	Issue Tracking System
LSI	Latent Semantic Indexing
NLP	Natural Language Processing
POS	Part of Speech Tagging
RE	Requirements Engineering
RQ	Research Question
SCS	Source Code Structure
SDK	Software Development Kit
SE	Software Engineering
SLR	Systematic Literature Review
TM	Trace Link Maintenance
TP	True Positive
UML	Unified Modelling Language
VCS	Version Control System
VSM	Vector Space Model

Part I

Preliminaries

Chapter 1

Introduction

This chapter begins in Section 1.1 with an introduction and explanation of the motivation as to why, for continuous trace link creation, maintenance and direct link usage, quality improvements of automatically generated trace links are essential. In Section 1.2 the problems which will be solved in this thesis are defined by two research goals. In Section 1.3 the contributions which address the research goals are listed. In Section 1.4 the design science research methodology which was used in this thesis is introduced, including a description of the design cycle performed in this thesis. In Section 1.5 the structure of the thesis is outlined. Section 1.6 concludes the chapter with a list of the publications regarding the thesis which have already been published as scientific work.

1.1 Motivation

Traceability in *Software Engineering* (*SE*) enables the tracing of artefacts by means of trace links between the artefacts [Gotel et al., 2012a]. Trace links are used between all kinds of *SE* artefacts such as requirements, source code, design artefacts and test cases, etc. [Cleland-Huang et al., 2014]. Out of the links between all these artefacts, trace links between requirements and source code are especially fundamental, since they enable the connection between the user point of view of a requirement and its actual implementation [Gotel et al., 2012b]. Furthermore, these kind of links also enable the tracing of requirements' evolution history which is an essential concern for *Requirements Engineering* (*RE*) [Gotel et al., 2012a].

In general, traceability between source code and requirements is important for many *SE* and *RE* tasks, such as maintenance, program comprehension, verification, etc. [De Lucia et al., 2010, Mäder and Egyed, 2011, Bavota et al., 2012, Bouillon et al., 2013, Rempel et al., 2014, Mäder and Egyed, 2015]. It is also a major concern of *SE* and *RE* research [Gotel et al., 2012b, Borg et al., 2014, Cleland-Huang et al., 2014, Panichella et al., 2015].

The manual creation of trace links is cumbersome and time-consuming and is not performed at all in real world projects [Gotel et al., 2012a]. Automatic trace link recovery and creation methods try to countervail the problem of the additional effort which is consumed in manual trace link creation.

Information Retrieval (*IR*)-based trace link creation is the most commonly used technique to automatically discover and create links between artefacts [Borg et al., 2014]. *IR* is used to create links between artefacts with a textual similarity. Thus, *IR*-based trace link creation is limited with respect to the similarity measure and the *IR* technique used [McMillan et al., 2009]. Further *IR*-based trace link creation focuses on a 'once-at-a-time' batch-oriented creation of links using structural requirements, such as use cases and to use the resulting links for verification purposes [De Lucia et al., 2008, Cleland-Huang et al., 2014]. Thus, research concerning *IR*-based trace link creation methods focuses on finding all existing links, for the purposes of recall optimization. The burden of wrong links and the efforts to filter out wrong links from a list of link candidates is accepted, since the list of link candidates is only created and processed once in a while [Briand et al., 2014, Niu et al., 2014].

However, in many companies, requirements are managed in *Issue Tracking System* (*ITS*) [Maalej et al., 2014a]. For open source projects, *Issue Tracking Systems* (*ITSs*) are even the de facto standard for all *RE* activities [Ernst and Murphy, 2012, Merten et al., 2016a]. In *ITS* the requirement's text is unstructured, since *ITSs* are used for many purposes, e.g. development task and bug tracking in addition to *Requirements Engineering* (*RE*). The mostly unstructured form of requirements in *ITSs* and the usage of *ITSs* for different purposes impair the results of *IR*-based trace link creation approaches in such environments [Merten et al., 2016b]. Furthermore, direct availability and usage of trace links during projects improves the performance of developers [Mäder and Egyed, 2015]. For ongoing creation and direct usage of links, wrong links and manual filtering of link candidate lists is not practical.

Once links have been created, they have to be maintained along with the changing of linked artefacts during the progression of a project [Wohlrab et al., 2016, Maro et al., 2016]. *Trace Link Maintenance* (*TM*) is a crucial part of the complete trace link management process. Without keeping trace links up to date, along with changes in the linked artefacts, trace links become obsolete and useless [Gotel et al., 2012a, Cleland-Huang et al., 2014]. A simple approach to *TM* would be to remove all existing links and simply to regenerate the links in the same way as in the initial generation.

However, this simple *TM* approach is often not practical due to the resources required for a complete regeneration of links. On the one hand, such resources are the required computing power, and, on the other hand, one has to account for the time spent and knowledge required for manually vetting links. For continuous creation and usage of links, the complete regeneration of links is even more impractical. Therefore, it is important to enhance trace link approaches with maintenance.

Thus, this thesis proposes a new automatic trace link creation approach for continuous link creation and the maintenance of links along with changing artefacts, which enables the direct usage of the created links.

1.2 Problem Statement

The problem of established trace link creation techniques for continuous link creation and usage is their insufficient quality regarding precision and recall [Cleland-Huang et al., 2014]. According to Gotel et al. [2012b], there is a trade-off between precision and recall: if *IR*-based methods have a good recall (up to 90%), the generated candidate links include a lot of *False Positives* (*FPs*), in which precision is between 10% to 20% [Gotel et al., 2012b]. Even with recall values of 90%, important links might be missing.

The reason why a 100% recall with *IR* is almost impossible is that related artefacts do not always share the same terms and thus link recovery is not possible. In the context of unstructured requirements in an *ITS*, the term mismatch problem is even more relevant than in a traditionally structured requirement context with restricted language. The primary reason for bad precision of *IR*-based trace link creation techniques is the dependency between precision and recall when using *IR*-based recovery techniques. To obtain high recall values, the threshold which defines that two artefacts are linked has to be low [Niu et al., 2014]. This often results in a bad precision, since many unrelated artefacts are included.

The maintenance of created links along with the changing of linked artefacts during the progress of a project is most often not considered at all in existing automatic trace link creation approaches. If maintenance is discussed in many cases, these approaches favour the removing of all existing links and completely regenerating the links.

In summary, existing *IR*-based trace link creation approaches have particular problems with unstructured requirements and with precision improvements that do not affect the recall. This results in impracticality concerning such approaches for continuous link creation, usage and maintenance. On the one hand, created link candidates require a manual assessment in order to be usable, and on the other hand link maintenance is only performed by a complete regeneration of links. Thus, this thesis targets two primary research goals related to the continuous creation, maintenance and usage of links:

G1 Design and evaluate an approach to improve the quality of automatically generated trace links, between unstructured requirements and source code, in comparison to existing approaches. The focus is to achieve a perfect precision for automatically and continuously generated links, which are directly usable during development, but which still keep the recall acceptable.

G2 Discover capabilities to maintain existing trace links along with the changes on linked artefacts to keep good link quality and extend the approach of **G1** with the capabilities discovered.

1.3 Contributions

To achieve these two research goals, this thesis presents the *Interaction Log Recording-based Trace Link Creation* (*ILog*) approach, which uses interaction logs and existing links as data sources for trace link creation and maintenance. The abbreviation *ILog* refers to the data source utilized ***I**nteraction **Log** Recording-based Trace Link Creation*. Interaction log recordings are the recordings of a developer's interactions from the artefacts' a developer has touched in an *Integrated Development Environment* (*IDE*). The usage of *ILog* targets a development set-up in which an *ITS* is used to manage requirements in the form of unstructured issues.

A core motivation to use interactions instead of the contents of linked artefacts was to avoid the problems concerning the bad precision of other link creation approaches, which rely on the contents of linked artefacts [Borg et al., 2014]. Generally speaking, the benefit of using developers' interactions instead of the contents of linked artefact is that developers are intelligent in comparison to an *IR*-based or other approach. It is likely that developers know what to do, i.e. they know which source code files are required when they work on a requirement, and that this results in a fewer number of wrong links compared to *IR* and other approaches.

The usage of interaction data in *ILog* was further motivated by different *SE* and *RE* research fields, in which interactions turned out to be a valuable data source to discover previously hidden knowledge.

Two such examples for interaction usage by others are the approach of Fritz et al. [2014], in which interactions are used to model the knowledge of developers and the approach of Konôpka and Bieliková [2015], in which implicit source code dependencies are discovered by analysing the interaction logs of developers. In both cases interactions were used to create new relations between previously unrelated entities, which basically represents the same problem which automatic trace link creation also tries to solve. Interactions and existing links as data sources can improve recall, since they are likely to yield new links compared to *IR* [Konôpka and Bieliková, 2015] and precision, since *FP* links are more unlikely for interaction logs and existing links than they are for *IR* [Soh et al., 2018].

In the context of the *ILog* approach, this thesis provides three contributions to improve the automatic creation (**G1**) and maintenance (**G2**) of trace links between requirements and source code.

The first contribution is the *ILog* approach itself, which is based on the usage of interactions on source code files, as captured within an *IDE*, while a developer is working on the implementation of a requirement. Furthermore, *ILog* also avoids

additional efforts for developers by using developers' work habits to assign recorder interactions to requirements. For the actual trace links creation, interactions are aggregated in a configurable manner by using the data recorded in the interactions. Finally, *ILog* enables the further improvement of the initially created links by utilizing existing links and aggregated interaction data.

The second contribution are three evaluation studies which have been conducted along with the development of the *ILog* approach. The result of these studies show that *ILog* can create trace links with precision to enable the direct usage of links, and that *ILog* is superior to other trace link creation approaches.

The third contribution is a *Systematic Literature Review* (*SLR*) which provides an overview of the state of the art of *Trace Link Maintenance* (*TM*). The *TM* approaches identified in the *SLR* are described by means of a standardized *TM* process and were assets with regard to their integration in *ILog*. Finally, based on the assessment, a *TM* process for *ILog* is sketched using the previously identified *TM* capabilities.

1.4 Research Methodology

The research conducted for this thesis follows the principles of the design science methodology as described by Wieringa [2014]. Design science is performed in design science projects [Wieringa and Moralı, 2012]. A design science project consists of two general consecutive activities. First, an artefact to improve something for a stakeholder is created. Second, the created artefact is empirically evaluated in a given context. To start a design science research project, it is necessary to specify a research goal. Since a design science project iterates over designing and evaluating, its research goals can be refined into design and knowledge goals.

Transferring these design science principles to the investigated research goal of this thesis, **G1** is a design goal and **G2** comprises aspects of a knowledge (the *SLR* about *TM*) and a design goal (integration of *TM* capabilities in *ILog*).

According to Wieringa [2014], the general design task of a design science project consists of three more specific tasks, namely (1) problem investigation, (2) treatment design and (3) treatment validation. These three tasks are called the design cycle, since they can be iterated multiple times in a research project. The goal of the problem investigation (PI) task is to establish the phenomena which must be improved, including a rationale as to why the improvement is necessary. In the treatment design (TD) task, artefacts are designed which could treat the problem. The TD task is followed by the treatment validation task (TV). In the TV task, the designed artefacts are evaluated to validate whether the problems of the PI task could be solved with the designed artefacts. Thus, all studies presented in this thesis are called evaluation studies.

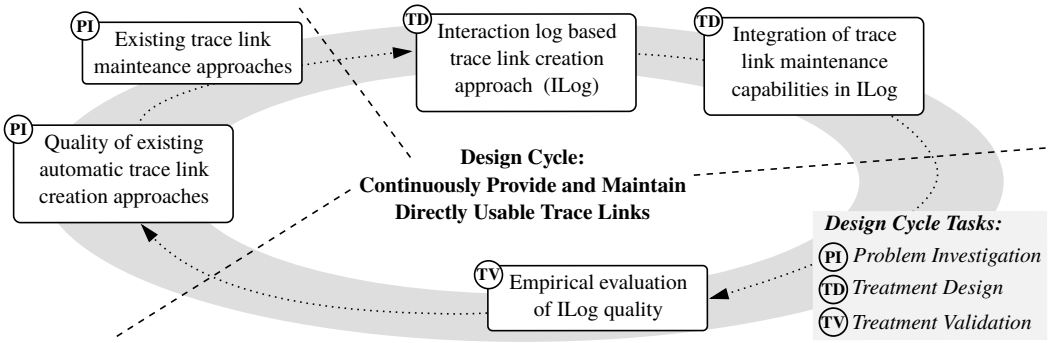

FIGURE 1.1. DESIGN CYCLE OF THE THESIS

Figure 1.1 shows the design cycle to investigate the continuous creation, maintenance and usage of automatically generated trace links performed in this thesis. In the following, the PI, TD and TV tasks performed throughout the design cycle are introduced in more detail. In the design cycle a PI, TD and TV tasks for continuous link creation (**G1**) and PI and TD task for link maintenance (**G2**) were performed.

For **G1** the design cycle starts with the first and general PI task in which existing automatic trace link creation approaches are reviewed with regard to their link creation techniques and quality. For **G2** the PI task is to investigate and assess existing *TM* approaches. Here, the focus is to identify *TM* approaches which maintain links along with changing artefacts during a project.

In the TD task for **G1** the *ILog* approach is introduced. For **G2** in the TD task maintenance capabilities of the previously investigated existing *TM* approaches have been assessed and a *TM* process with selected *TM* capabilities for *ILog* is presented.

In the TV task for **G1** the *ILog* approach has been evaluated in three empirical studies. For **G2** no TV task was performed. However, potential evaluations for *ILog* with *TM* capabilities are discussed in the concluding part of the thesis.

1.5 Structure of the Thesis

Table 1.1 shows an overview of the structure of the thesis and the performed design cycle. For the introduction and conclusion parts, the chapters and links to the chapters are shown. For the parts comprising a task of the design cycle, the table shows the investigated *Research Questions* (*RQs*), which address the previously stated research goals **G1** and **G2**, the result and links to the respective chapters.

Part I presents the preliminaries which includes this introduction and the fundamentals required for the rest of the thesis. Part II presents the PI tasks for **G1**, which consists of a review of existing trace link creation approaches and **G2**, which presents a *SLR* for *TM* approaches. Part III presents the TD tasks for **G1**, the *ILog* approach, and for **G2**, a sketch of how to integrated selected found *TM* capabilities in *ILog*. Part IV presents the TV task for **G1**, which consists of three evaluation

studies for the *ILog* approach, including an initial introduction of the used projects, data and tools and a final discussion of the overall results. Part V presents the conclusion of the thesis, including a summary and an outlook of future work.

Table 1.1. Structure of the Thesis

		Preliminaries	Chapter
	Part I	Introduction	1
		Fundamentals	2
		Problem Investigation	
	Part II	***Quality of existing automatic trace link creation approaches*** *RQ*: Which automatic trace link creation approaches exist and what is the quality of the resulting links? Result: Structured description of existing automatic trace link creation approaches and their quality	3
		Maintenance of existing trace links *RQ*: Which approaches exist for *Trace Link Maintenance* (*TM*)? Result: Structured description and assessment of *TM* approaches	4
		Treatment Design	
Design Cycle	**Part III**	***Interaction Log Recording-based Trace Link Creation (ILog) approach*** Result: *ILog* approach, consisting of: (1) Interaction log recording with: • manual assignment • *commit-based* assignment (2) Trace link creation by interaction aggregation (3) Trace link optimization techniques • precision optimization (wrong link detections by using interaction data and *Source Code Structure* (*SCS*)) • recall optimization (by *SCS*)	5
		***Integration of trace link maintenance in* ILog** *RQ*: To what extent can existing *TM* approaches be transferred to *ILog*? Result: Extension of *ILog* with capabilities from existing *TM* approaches	6
		Treatment Validation	
	Part IV	***Evaluation of* ILog *quality*** *RQ*: What is the quality of trace links generated with *ILog*? Results: Three empirical evaluation studies which show and discuss the good quality of links created with *ILog*	7, 8, 9, 10, 11
		Conclusion	
	Part V	Summary and Outlook	12, 13

1.6 Previous Publications

Parts of this thesis have already been published as scientific work. Table 1.2 provides an overview of these publications in chronological order, including a reference to the corresponding chapters of the thesis.

Table 1.2. Publications in the Context of the Thesis

No.	Publication	Chapter
[IL1]	Paul Hübner. Quality Improvements for Trace Links between Source Code and Requirements. In *Proceedings of the REFSQ Workshops, Doctoral Symposium, Research Method Track, and Poster Track*, volume 1564, Gothenburg, Sweden, 2016. CEUR-WS	1, 5
[IL2]	Paul Hübner and Barbara Paech. Using Interaction Data for Continuous Creation of Trace Links Between Source Code and Requirements in Issue Tracking Systems. In *Proceedings of the 23rd International Working Conference - Requirements Engineering: Foundation for Software Quality (REFSQ)*, volume 10153 of *Lecture Notes in Computer Science (LNCS)*, pages 291–307, Essen, Germany, 2017. Springer	5, 7, 8, 11
[IL3]	Paul Hübner and Barbara Paech. Evaluation of Techniques to Detect Wrong Interaction Based Trace Links. In *Proceedings of the 24th International Working Conference - Requirements Engineering: Foundation for Software Quality (REFSQ)*, volume 10753 of *Lecture Notes in Computer Science (LNCS)*, pages 75–91, Utrecht, The Netherlands, 2018. Springer	5, 7, 9, 11
[IL4]	Paul Hübner and Barbara Paech. Increasing Precision of Automatically Generated Trace Links. In *Proceedings of the 25th International Working Conference - Requirements Engineering: Foundation for Software Quality (REFSQ)*, volume 11412 of *Lecture Notes in Computer Science (LNCS)*, pages 73–89, Essen, Germany, 2019. Springer	5, 7, 10, 11
[IL5]	Paul Hübner and Barbara Paech. Interaction-based Creation and Maintenance of Continuously Usable Trace Links. *Empirical Software Engineering*, 2020 *status: submitted*	4, 5, 6, 10, 11

Chapter 2

Fundamentals

This chapter introduces the fundamentals for this thesis. In Section 2.1 the data sources which are used for trace link creation and which are relevant in the context of this thesis are introduced. This includes *Issue Tracking Systems* (*ITSs*), *Version Control Systems* (*VCSs*), *Source Code Structure* (*SCS*) and interaction data. In Section 2.2 the fundamentals of traceability, i.e. the automatic creation of trace links, *IR* and other techniques which utilize the previously introduced data source for trace link creation are introduced. In Section 2.3 the measurement fundamentals to evaluate and rate the quality of a trace link creation approach are introduced. This includes the concept of a gold standard and the definitions for precision, recall and further quality-rating measures for created links.

2.1 Data Sources

This section introduces the data sources used for trace link creation in this thesis.

2.1.1 Issue Tracking Systems

Nowadays, many software companies use *Issue Tracking System* (*ITS*) to specify their requirements [Maalej et al., 2014a]. For open source projects, the usage of an *ITS* is a crucial point and a de facto standard [Merten et al., 2016a]. In *ITS*, the requirements text is unstructured and requirement issues are mixed with other issues for e.g. bug tracking, task and test management [Merten et al., 2016a].

The unstructured requirements specification such as those which are used in *ITS* is contrary to the structured requirements specification, in which the requirements follow a more strictly predefined template. A sample for a stricter kind of format is a use case. In a use case a requirement is described in a sequence of actions performed by different actors, including data transferred between the actions. Further such structure formats often restrict the language used. Textual content-based trace link creation methods such as *IR* can use structure information, e.g. for specific

preprocessing, and benefit from a restricted language in more significance for the similarity measures [Hayes et al., 2006].

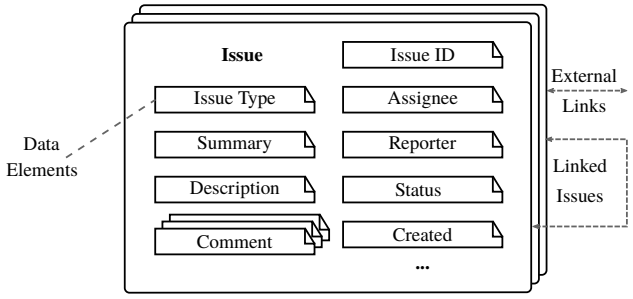

Figure 2.1. Typical Issue Data Elements Relevant for Traceability

Figure 2.1 shows the typical data elements of issues which are managed in an *ITS*. The *issue type* is used to indicate the purpose of an issue, e.g. *Bug* for bug tracking, *task* or *sub task* for a task to be performed, *story*, *feature* or *epic* for requirement specification, *test case* for testing, etc. A unique *issue ID* is used to reference an issue. This can take place in the *ITS* typically by other issues. For example, when a requirement issue is linked to an task issue and to a test case issue, the task issue specifies how to implement the requirement and the test case issue specifies how to perform the testing for the requirement. External links target the elements in other systems, e.g. pages in a Wiki or commits in a *VCS* (cf. the following Sections 2.1.2 and 2.2.3), etc. The *summary* and *description* are unstructured text which summarize and describe the details of the issue. Comments enable users to discuss an issue and are also often used to clarify concerns and add more details. For content-based trace link creation methods such as *IR*, the summary, description and comment data elements are most often used [Merten et al., 2016a]. Further typical data elements are an *assignee*, i.e. the user who is assigned to process the issues, a *reporter*, i.e. the user who initially created the issue, the processing *status* of the issue, a *creation* date, etc. In the context of this thesis, the content elements of the issue (summary, description and comments) are relevant for *IR*-based trace link creation and the usage of issue IDs in *VCS* commit messages is relevant for commit-based trace link creation and *ILog*.

2.1.2 Version Control Systems

A *Version Control System* (*VCS*) is used in software development to manage the changes made to artefacts during a project. *VCS* are designed to handle changes between textual artefacts and thus are mainly used in projects for implementation artefacts such as source code files. In the following the *VCS*-specific data and mechanisms that are used for trace link creation in this thesis are introduced.

Figure 2.2 shows an overview of data elements and presents how they are created in an *VCS*. Artefacts such as source code files are managed in a *repository*. In a

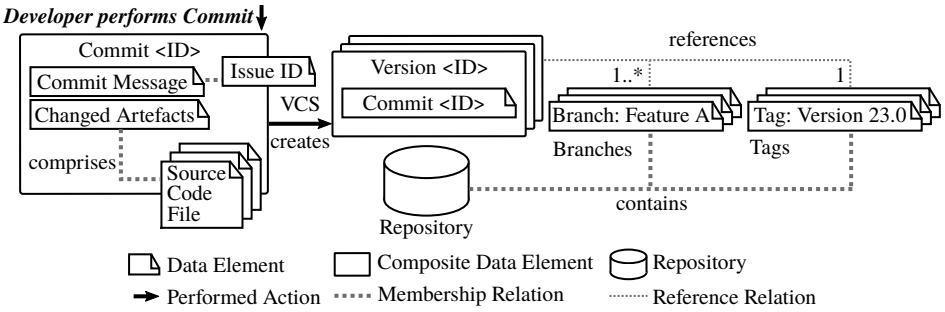

FIGURE 2.2. VERSION CONTROL SYSTEM DATA ELEMENTS RELEVANT FOR TRACEABILITY

centralized *VCS* there is only one repository. In a decentralized *VCS* there can be multiple repositories, in this case typically one local repository for every developer and a centralized repository to synchronize all changes are used.

After the artefacts have been changed by a developer, the developer performs a *commit* to the repository. The commit includes all *artefacts changed* since the last commit. Further to each commit a developer has to provide a *commit message*. A common convention and a de facto standard for such a commit message is to provide the issue identifier (ID), or even multiple issue IDs, from the *ITS* which the commit refers to and a short textual description of the performed changes. For each performed commit, the *VCS* creates a new *version* of the artefacts in the repository.

As shown in Figure 2.2, a repository comprises *branches*. Branches are created by developers to work independently of each other. Thus, a branch is used in order to disconnect a series of commits (versions) from other changes, e.g. to implement a completely new feature. Typically the changes performed in one branch are merged back to other branches after a while, e.g. if the implementation of a new feature is finished. A main branch is used to mark the current stable state of artefacts in the repository which is created from merging individual branches to this main branch. A *tag* is a text label-marking of a certain version created by a commit, in order to make the version easily able to be referenced. Tags are stored in the repository. Tagging is often used during a release to mark the version of artefacts used in the release.

LISTING 2.1. EXAMPLE VCS COMMIT MESSAGE

```
commit 92a8f249e6b6eda2cb497156fc4927b6b86b1859
Author: dev1 <dev1@uni-heidelberg.de>
Date:   Thu Feb 16 15:40:11 2017 +0100

ISE2016-224 tooltip slider added
```

Listing 2.1 shows an example of a typical commit message which refers to the *ITS* issue with issue ID *ISE2016-224*. In addition, the listing shows above the commit message that the commit has a unique ID, an author, and a date time stamp.

Common *VCS*, such as Git[1] provide much more functionality than that which is described here. A complete overview of Git's functionality can be found in [Chacon and Straub, 2014].

2.1.3 Source Code Structure

Source Code Structure (*SCS*) are the different kinds of relations between source code files. In the context of this thesis *SCS* denotes the call and data dependencies between code files and classes [Kuang et al., 2015]. Using the code structure to improve trace link creation is part of traceability research.

Kuang et al. [2017] created a distance measure between different code files which are based on the *SCS* between the code files, and they used this distance measure to improve trace links which were initially created with *IR*. Ghabi and Egyed [2012] created a set of patterns based on *SCS* not only to judge existing trace links to requirements about their correctness but also to detect missing links. Rahimi et al. [2016] used the *SCS* to detect certain refactorings and then used a set of refactoring-specific rules to create and maintain links to requirements. *SCS* is also used in the *ILog* approach of this thesis to improve initially created links.

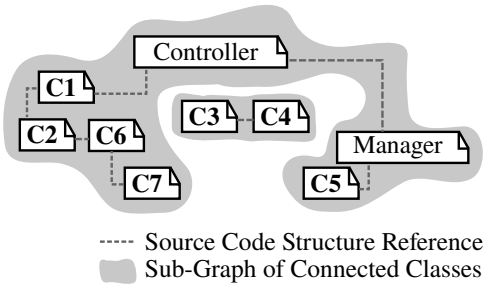

Figure 2.3. Example Source Code Structure between Classes

Figure 2.3 shows an example *SCS* of the classes *Controller, Manager* and *C1* to *C7*. In the following this example is used to illustrate the different relevant *SCS* concepts and relations. Based on the references between the classes, the classes can be separated into two *sub-graphs*, in which the classes are the nodes and the references are the edges of the graph. The first smaller sub-graph consists of the two classes *C3 and C4*. The second larger sub-graph consists of the other classes *C1, C2, C5, C6, C7, Controller* and *Manager*.

In the following code listings, with code excerpts from the second sub-graph, classes are used to illustrate how different relation types between classes are used to create the *SCS*, as shown in Figure 2.3. The code listings are presented in a Java-oriented syntax with parts which are relevant for relations, and thus *SCS*, highlighted.

[1] `https://git-scm.com/`, Git *VCS* website

Listing 2.2. *SCS* Example Class Controller

```
1  public class Controller{
2    Manager manager = <init Manager>;
3    public void control(C1 c1){
4      c1.init();
5      manager.manage();
6      c1.close();
7      ...
```

Listing 2.2 shows the code excerpt of the *Controller* class. In line 2 a reference to the class *Manager* is created by creating and initializing a new *Manager* instance. In line 3 a reference to *C1* is created by using an instance of *C1* as input parameter of the method *control()*. In lines 4 to 6 further references are created by method calls.

Listing 2.3. *SCS* Example Class Manager

```
1  public class Manager implements C5 {
2    public void manage(){
3      ...
```

Listing 2.3 shows the code excerpt of the class *Manager*. In line 1 a reference to the class *C5* is created by implementing the interface defined in *C5*.

Listing 2.4. *SCS* Example Class C1

```
1  public class C1 extends C2 {
2      ...
```

Listing 2.4 shows the code excerpt of the class *C1*. In line 2 a reference to the class *C2* s created by extending *C1* by inheritance of the implementation of *C2*.

To summarize the relations when a class references other classes in its attributes or by method calls, a class implements an interface or a class extends another class are used to created the *SCS*. In addition a set of classes connected by *SCS* is denoted as a *sub-graph*.

2.1.4 Interaction Data

In this section the basics of interaction data as required for the *ILog* approach are introduced. After this, an overview of interaction data usage by others is given.

2.1.4.1 Interactions of Developers

In the context of this thesis and the *ILog* approach, all interactions of developers with artefacts during the usage of an *IDE* are considered. The intention of the usage of an *IDE* by a developer is to perform a task. Such a task can be the implementation of a requirement or bug fixing etc. Most of the artefacts which are interacted with during such a task are source code files. Since the goal of this thesis is to create links between source code and requirements, references to interactions during the

thesis always refer to the context of a developer who is working with an *IDE* on a requirement-related implementation task and who is interacting with source code files during this task. Interactions recorded during a period of time and stored in a place, such as in the workspace of the *IDE*, are called *interaction log recordings*.

Common *IDEs* such as Eclipse[2] provide the functionality to record developer's interactions [Murphy et al., 2006]. A interaction of a developer which is performed in an *IDE* creates a *interaction event*. A single recorded interaction event comprises multiple data attributes. Common attributes are a *time stamp* of the interaction occurrence, the *source code file* which has been interacted with, the *type of interaction* (e.g. select, edit, etc. as well as more fine-grained types such as keystroke, menu selection, etc.) and future information, such as the *part of the IDE* in which the interaction occurred (e.g. navigator, editor, main menu, etc.), developer-specific information such as a *user name*, etc.

Based on the interaction data described, an interaction, when it is recorded in an interaction event, and when it touches the implementation artefact, i.e. source code file, I while working on requirement R, can be used to create a trace link between I and R.

2.1.4.2 Usage of Interaction Data

This section reports on the use of the interaction data of developers by other researchers. The focus for the selected approaches is the usage of interaction data to create relations (i.e. links) between artefacts. The intention was to identify principles and techniques which can be transferred to the *ILog* approach developed in this thesis. The approaches were identified by an initial exploratory search, a review of the *Mining Software Repositories*[3] conference proceedings[4] from 2004 to 2016, and by searching for related work when performing the evaluation studies of *ILog*.

Maalej et al. [2014b] describe how to use interaction data, based on examples of existing recommendation systems, in order to trigger recommendations about the source code artefacts used. Furthermore, they describe different methods to aggregate this data to sessions, tasks, and activities, and show how to filter such data for productive use.

Maalej and Ellmann [2015] conducted a study on the similarity of development tasks using the context of the tasks. In their study they used Mylyn[5] for the collection of task data. Thus, for the definition of the task context, they also used the Mylyn's degree of interest (DOI) model. In summary the DOI model is based

[2] `http://www.eclipse.org`, Eclipse *IDE* website

[3] `http://www.msrconf.org/`, *International Conference on Mining Software Repositories* Website

[4] `https://dblp.uni-trier.de/db/conf/msr/`, conference proceedings

[5] Mylyn is a plug-in for the Eclipse *IDE* which records interactions in order to support the implementation tasks of developers. Mylyn-recorded interactions have also been used in this thesis. With this in mind, Mylyn is introduced in further detail in Chapter 7 of the *evaluation part* evaluation part of the thesis.

on the number of interactions with files and the type of interactions (edit, select, etc.) and a rating which is calculated on the basis of those two values. The results of their study showed that their file interaction-based task context model achieves similar results as when using *IR* and the textual similarity of task descriptions for finding similar development tasks. However, their approach has the advantage of not requiring a textual task description which is often miss.

Cleland-Huang et al. [2012] present a study in which they implemented and evaluated an approach to recommend trace links between the system model (specified in *Unified Modelling Language* (*UML*)) and requirements during model changes. They use edit interactions to trigger recommendations of trace links. However, for the creation of trace links, *IR* is used in their approach.

Omoronyia et al. [2009] published an approach in which interactions between source code and structured requirements which were specified as use cases are captured and used to visualize and navigate trace links. Their tool support focuses on visualizing the trace links after a task has been performed and not on the direct availability and usage of trace links. In a follow-up paper [Omoronyia et al., 2011] by the same authors, they also use interactions for trace link creation. In it they consider developer collaboration and rank interaction events. Their approach achieves a precision of 77% in the best case.

The use of interaction data to find software artefact relations is also applied in the domain of software architecture. Konôpka and Navrat [2015] find relations between source code artefacts and tasks based on interaction data recorded in an *IDE*. The recorded interaction data is used for grouping code, but not for trace link creation. [Konôpka and Bieliková, 2015] is another paper by the same authors. In this they also use interaction logs not only for grouping but also in order to detect implicit relations between code files.

Soh et al. [2018] showed through an observation study, along with interactions recorded using Mylyn, that observed interaction durations do not always correspond to recorded interaction durations. More precisely, they show that the assumptions that the time which is recorded for an interaction is the time spent on a task, and that an edit event which is recorded by Mylyn corresponds to modification in the code, are not true. They were able to detect these differences by comparing the interactions and video-capturing of developer behavior in a quasi-experiment. These differences are not due to any misbehavior of the developers, but only due to Mylyn's recording algorithm. For example, searching and scrolling is not counted in the time spent and the idle time is not treated correctly.

El-Ramly and Stroulia [2004] define a process that supports the detection of interaction patterns during the usage of a computer program by a user interface. The interaction patterns can be used to detect spurious events within a task which is performed by a user. As a result of this, the user interface can be optimized to avoid the spurious events.

Schneider et al. [2004] discuss the analysis of local interaction histories of a *VCS*. Their core idea is that the local interaction histories comprise valuable information which is not present in a shared *VCS* repository. Among other applications, they suggest using the file search and browsing patterns of developers to identify relations to files and to identify performed refactorings.

Parnin et al. [2006] suggest augmenting the revision history of a *VCS* with the interaction history of programmers. In particular, they suggest adding *program viewing and editing history* with all historical artefacts within a *VCS*. They performed an evaluation in which a *IDE* plug-in was used to record click, navigation, move, and edit interactions of ten developers while they were working on different projects. After that, the recorded interactions were analysed. A key finding was that interaction data contains previously unavailable data and that applications such as recommender systems and developer behaviour analyses can benefit from this data.

Rastkar and Murphy [2009] compared the change-sets of commits to recorded interactions for describing software evolution tasks. In contrast to interactions, a change-set only covers the final result, i.e. the changes in artefacts, of such a task. In the performed evaluation study Rastkar and Murphy analysed change-sets and recorded interactions to find similar bug reports. They found that the two methods produce very different connections, but neither is clearly superior to the other. They conclude that both methods cover different aspects.

Robbes and Rothlisberger [2013] used interaction data which was recorded in the Mylyn project to create metrics in order to rate the expertise of developers. Such an expertise metric can be used to assign tasks to developers efficiently. Interaction data was used for a task-specific *required time spend* and a *number of interactions performed* metric. The assumption for expertise rating stated that a developer who is familiar with the relevant code entities will require less time and slightly fewer interactions to complete a task. However, the achieved evaluation results were uneven and the authors suggest that further investigation is necessary.

Zanjani et al. [2014] used interaction data to improve the impact analyses for change request. In contrast to using a snapshot of the entire source code or the commits associated with the relevant change request for impact analysis, the presented approach uses all revision of source code which has been committed or interacted with while performing a change request. Then, *Natural Language Processing* (*NLP*) and *IR* are used to find similarities for new change request in the history of performed change requests. This approach showed an increase in accuracy compared to previous approaches.

As a follow-up work to [Zanjani et al., 2014], Zanjani et al. [2015] describe an approach to find the most suitable developer to perform a change request. First, the textual similarities of a change request with source code files are used to identify source code files which are relevant for the change request. Then, the interactions of developers with source code files, which are recorded with Mylyn while performing

other change requests, are used to identify suitable developers. The number of interactions with a source code file and the relative time which is taken in working on this source code file compared to others determined the developer's expertise, resulting in a ranked list of developers for each incoming change request.

In summary the presented usage of interaction data, it can be seen that different tasks such as recommending artefacts, grouping artefact and also creating relations between artefacts, are supported. The interaction data-based support in these tasks is used to discover previously hidden or only implicit information, e.g. groups of artefacts, knowledge of a software developer about artefacts, etc. Further the usage of a *VCS* in conjunction with interactions is prominent in these approaches.

The approaches of Omoronyia et al. [2011] and Konôpka and Bieliková [2015] explicitly use interactions for trace link creation. However, instead of links between requirements and source code files, Konôpka and Bieliková only created links between different source code files, whereas Omoronyia et al., do not directly provide the resulting links for usage, but create a list of link candidates that requires further vetting of these links.

2.2 Traceability

The term traceability in software engineering is used to express the fact that artefacts which are created through different software engineering life cycle phases are traceable. Thus, traceability enables the tracing of an artefact's evolution history. That is to say, traceability makes links between artefacts from the same and different *Software Engineering* (*SE*) life cycle phases explicit, e.g. by linking requirements with other requirements or by linking requirements with the source code implementing the requirement, etc. [Gotel et al., 2012b]. Links between artefacts from the same software engineering life cycle phase, i.e. which are on the same abstraction level, are also considered as horizontal traceability, whereas links between artefacts from different software engineering life cycle phases, i.e. which are on different abstraction levels, are considered as vertical traceability [Cleland-Huang et al., 2014].

In the *ILog* approach developed in this thesis, vertical traceability by the automatic creation of trace links between source code and requirements is considered. Linking between requirements and source code is crucial, since it presents the connection from the customer's point of view of the requirements to their actual implementation, i.e. the source code [Cleland-Huang et al., 2014]. Manual trace link creation is cumbersome, error prone due to a lack of overall expert knowledge, and rarely performed at all. Thus, the automatic creation of trace links is an established *SE* and *RE* research field [De Lucia et al., 2011a].

The following section 2.2.1 introduces the concept of automatic trace links creation. Section 2.2.2 introduces *Information Retrieval* (*IR*) techniques, which are the most common techniques for automatic trace link creation. Thus, in the con-

text of this thesis *IR*-based trace link creation approaches serve as the reference for comparison and rating of the *ILog* approach. Further Section 2.2.3 outlines other techniques used for automatic trace link creation. These other techniques can be summarized as rule-based techniques [Cleland-Huang et al., 2014].

2.2.1 Automatic Trace Link Creation

Most automatic trace link creation techniques utilize the contents of linkable artefacts [Borg et al., 2014]. Commonly these techniques are called *Information Retrieval* (*IR*)-based techniques. As Borg et al. report in their review of *IR*-based trace link creation approaches, the term *Natural Language Processing* (*NLP*) is used for *IR*-based trace link creation techniques as well, even if they refer to *IR* techniques. Thus, this thesis follows the suggestion of Borg et al. [2014] to use only the term *IR* when referring to automatic trace link creation techniques which utilize the textual contents of linkable artefacts. Other techniques for automatic trace link creation can be summarized as rule-based [Rahimi and Cleland-Huang, 2018]. This also includes approaches which apply machine learning techniques [Rath et al., 2018].

2.2.2 Information Retrieval

In this section the basic concept of *IR*, as required for *IR*-based trace link creation, is introduced. *Information Retrieval* (*IR*) is a machine or computer-based search for information within a set of artefacts [Baeza-Yates and Ribeiro, 2011]. The core assumption for *IR*-based trace link creation is that artefacts which are textually similar are related with each other and should be linked. Studies have shown that even different kinds of software engineering artefacts, such as documentation, requirements, source code, etc. use a similar terminology [Antoniol et al., 2002, Dekhtyar et al., 2004]. In *IR*-based trace link creation, *IR* techniques are used to measure the textual similarity of two artefacts. The textual similarity of two artefacts is then used to establish a link between the two artefacts if the measured textual similarity is above a certain threshold. De Lucia et al. [2011a] state that the process of trace link recovery consists of four key steps:

1. document parsing, extraction, and preprocessing
2. corpus indexing with an *IR* technique
3. ranked list generation
4. analysis of candidate links

In the following the further principles for *IR*-based trace link creation are introduced with respect to these four steps.

2.2.2.1 Preprocessing

To make the application of *IR* techniques as reasonable as possible, first it is necessary to preprocess the textual contents of artefacts. The preprocessing of textual artefacts is essential for the later application of the *IR* techniques. Typically, preprocessing consists of several steps, i.e. the successive application of different preprocessing techniques. Some of them are fundamental and some are specific to the artefacts and data sources used.

Stop word removal, punctuation character removal and stemming are common preprocessing steps which are also often used in *IR*-based trace link creation [Manning et al., 2008, Baeza-Yates and Ribeiro, 2011, Borg et al., 2014]. Stop word removal removes common words which have no impact on the similarity of artefacts. Stop words can be articles and prepositions such as *the, a, of* etc. as well as typical key words of programming languages such as*public, void,* etc. Punctuation character removal is self-explanatory, i.e. the removal of characters such as *;!?*, etc. Stemming is the process of reducing words to their stem or root form, e.g. *driving* becomes *drive*. The de facto standard for the stemming of English language is the *Porter Stemmer algorithm* [Baeza-Yates and Ribeiro, 2011]. It is used in most *IR*-based trace link creation approaches and in all evaluations comprising *IR*-based trace link creation as performed in this thesis.

A source code specific preprocessing step is identifier splitting. In source code multiple words are often combined to a single identifier, e.g. *BugzillaTaskServiceManager*, whereas in requirements separate words are used [De Lucia et al., 2011b, Ali et al., 2011]. Thus performing identifier splitting in source code files can significantly improve the similarity of source code and requirements artefacts. Camel case identifier splitting is a specific kind of identifier splitting e.g. *BugzillaTask* becomes *Bugzilla Task*. Camel case notation is especially common in Java source code but also in source code which uses other programming languages. Since the source code files used in the evaluation studies of this thesis are either in Java or JavaScript, camel case identifier splitting has been applied in all performed evaluation studies. This preprocessing proceeding in the performed evaluations is also consistent with the suggestions of Borg et al. [2014]. This is because Borg et al. recommend that the application of text preprocessing techniques should always be artefact type-specific and that the application of different preprocessing techniques and their combinations should be evaluated for the best results of generated trace links.

2.2.2.2 Indexing

After the preprocessing of textual artefacts, the next step of an *IR* technique is to index the terms of the preprocessed textual artefacts. The goal of indexing is to represent the artefacts in an document space by extracting information about the occurrence of terms within the artefacts. The occurrence information of a term

is then used to define similarity measures between the artefacts. Finally, a set of source artefacts is compared to a set of target artefacts and the similarity measures are used to rank all possible pairs (of artefacts) by their similarities. These pairs of artefacts represent the trace link candidate list. Commonly this list requires further manual vetting of the link candidates by an expert [De Lucia et al., 2011a].

Terms extracted from documents (artefacts) are stored in a so-called *term-by-document matrix*. In this $m \times n$ matrix, m is the number of all unique terms that occur within the documents (artefacts) and n is the number of documents. An entry w_{ij} in this matrix is a measure of weight or relevance of the i_{th} term in the j_{th} document [Baeza-Yates and Ribeiro, 2011]. The three core measurements for the *term weight* formulation as used in *IR*-based trace link creation techniques are:

1. *Term Frequency* ($\mathrm{tf}(t,d)$): terms (t) that are repeated multiple times in a document (d) are considered as prominent.

2. *Document Frequency* ($\mathrm{df}(t)$): terms that appear in many documents are considered as common and not very indicative. The inverse document frequency ($\mathrm{idf}(t) = \log\left(\frac{n}{\mathrm{df}(t)}\right)$, in which n is the number of all documents) weighting method, is based on the document frequency *df*.

3. *Document Length*: to avoid side-effects of longer documents scoring higher, normalization is performed.

Term frequency-inverse document frequency (*tf-idf*) is the most common weighting and normalization function used for term frequency, document frequency and document length. It is used to weight a term, based on the document length and the frequency of a term, and thus it weights a term for a single document and all documents. It is defined as:

$$\begin{aligned}\text{tf-idf}(t,d) &= \mathrm{tf}(t,d) \cdot \mathrm{idf}(t) \\ &= \mathrm{tf}(t,d) \cdot \log\left(\frac{n}{\mathrm{df}(t)}\right)\end{aligned}$$

Further possible improvements to *tf-idf* could also be to normalize *tf* within *tf-idf*, so that the effect of multiple occurrence of a term in a document is limited as well [Baeza-Yates and Ribeiro, 2011].

2.2.2.3 Trace link Creation Techniques

With the term-by-document matrix representation, different *IR* techniques can be used to rank pairs of source and target artefacts based on their similarities. That is to say, *IR* techniques are used to execute search queries which aim to retrieve all relevant artefacts while minimizing the non-relevant artefacts [Baeza-Yates and Ribeiro, 2011]. When using *IR* techniques for trace link creation, the query concerns

the textual similarity between two artefacts. The most common *IR* techniques for trace link creation[6] are probabilistic models and vector space-based models [De Lucia et al., 2011a]. In a probabilistic model, a source artefact is ranked according to the probability of its being relevant to a particular target artificial. In vector space-based models, artefacts are represented as vectors of terms from the artefacts. In these source artefacts are ranked against target artefacts by computing a distance function between the vectors of the artefacts [De Lucia et al., 2011a]. Vector space-based models are the most used and most well-established trace link creation techniques [Borg et al., 2014]. The usage of probabilistic models is rare in current *IR*-based trace link creation approaches, since it is known that they perform worse on average in comparison to vector space-based models [Borg et al., 2014]. Probabilistic models are more common in other *IR* applications such as full text search engines [Baeza-Yates and Ribeiro, 2011]. Thus, in this thesis only vector space-based models are introduced and used for the comparison with the *ILog* approach. In the following the two most common vector space-based models *Vector Space Model* (*VSM*) and *Latent Semantic Indexing* (*LSI*) are introduced [Borg et al., 2014, Gotel et al., 2012b].

Vector Space Model (VSM)

In the most common vector space-based model, the cosine similarity between two term vectors (documents) is used as a measure for the textual similarity. The cosine similarity measures the similarity between the two term vectors which represent the artefacts, and which is based on the cosine of the angle between the term vectors. This technique is commonly referred to as *Vector Space Model* (*VSM*) [De Lucia et al., 2011a]. The numerical value of the cosine similarity is between 0 and 1 [Borg et al., 2014]. 0 indicates no similarity between two artefacts and 1 indicates that two artefacts are identical. In order to define whether two artefacts are related with each other and should be linked, a threshold value for the cosine similarity is used Cleland-Huang et al. [2007]. Thus, varying this threshold value also varies the number of created trace link candidates. An alternative to the cosine similarity, and thus a different vector space-based model, would be to use the inner product of the two terms vectors instead of the cosine [De Lucia et al., 2011a].

Latent Semantic Indexing (LSI)

Latent Semantic Indexing (*LSI*) is also a vector space-based model. The difference between *Vector Space Model* (*VSM*) and *LSI* lies in the way in which the term comparison is performed. Whereas *VSM* measures the similarity based on terms which are directly extracted from the documents, *LSI* measures the similarity based on concepts. Concepts are high level abstractions of the terms used and can been

[6]Sometimes trace link creation is also denoted as *trace link recovery* by authors, yet both refer to the same concept of initially creating links using linkable artefacts

seen as the topics of the artefacts [Deerwester et al., 1990, Baeza-Yates and Ribeiro, 2011]. Concepts cluster related terms with respect to documents and related documents with respect to terms [De Lucia et al., 2011a]. Thus, *LSI* enables similarity matches between artefacts which do not contain exactly the same terms and thus overcomes the polysemy and synonymy problems of *VSM*; e.g in *VSM* the terms *bicycle* and *road bike* are handled as different terms, whereas in *LSI* these two terms would match to the same concept.

2.2.2.4 Link Candidate Processing

In the final two steps of the trace link recovery process of De Lucia et al. [2011a], the resulting link candidates are first ranked according their textual similarity. As previously introduced, the cosine similarity is used as a textual similarity measure of two artefacts representing a link candidate. Furthermore, the link candidate list is then processed, most often manually, to remove *False Positives (FP)*. Since it is known that *IR* methods produce a lot of *FP* links, the processing of *IR*-created link candidates lists is even a research subject of its own [Hayes et al., 2006, Niu and Mahmoud, 2012, Falessi et al., 2017]. Simple strategies for processing the link candidates list are *constant cut point* and *variable cut point*. In the *constant cut point* strategy, link candidates are only considered if their textual similarity is above a certain threshold. In the *variable cut point* strategy, the list is processed until a certain percentage of the candidate links have been considered, since it is known that links later in the list (with lower similarity) are more likely to be wrong.

2.2.3 Commit-based and Further Trace Link Creation Techniques

Gotel et al. [2012b] summarize the other trace link creation techniques as *rule-based*. Rule-based techniques often utilize properties of artefacts in combination with *IR*, e.g. by creating links between all requirements and the activities of an activity diagram whose name is textually similar to it. The rule-based techniques also comprise machine learning-based techniques. Machine learning techniques are the core techniques for data mining which is in turn the principle of discovering new knowledge from existing data [Witten et al., 2016]. Basically machine learning uses a training data set, which contains correctly classified data, e.g. linked artefacts and trace links between the artefacts, in order to train a classifier. Later the trained classifier is then used to classify new data, e.g. to create trace links between new artefacts. Furthermore, there is a differentiation between supervised and unsupervised learning. In supervised learning there is a teacher signal and the results created by a classifier can be compared to the teacher and handled accordingly, i.e. they can be accepted or rejected. In unsupervised learning no teacher exists and the results of a classifier can only be assessed manually or with further other techniques.

Commit-based trace link creation is a common rule-based trace link creation technique. It uses issue identifiers (IDs) which are provided by developers in *VCS* commit messages to link all files affected by a commit to the issue which is specified by the issue ID (cf. Section 2.1.2). The usage of issue identifiers (IDs) to link commits to requirements and bug reports is a common convention in open source projects [Bird et al., 2009, Merten et al., 2016a, Rath et al., 2018]. Developers often use this principle to create trace links themselves [Rath et al., 2017].

Rath et al. [2017] report a dataset *Ilm7* which they created from seven open source projects for the purpose of evaluating traceability research. They used commit-based trace link creation, i.e. the issue IDs in commit messages, in order to link issues to code files. They report that only 60% of the commits contain an issue ID.

In their follow-up work [Rath et al., 2018], they use the *Ilm7* dataset to train different machine learning classifiers to countervail the problem of commits without issue IDs. To train their classifiers, they not only used the files and issue IDs from commits, but also the textual similarity (*IR*) between different artefacts (i.e. the commit message text, the issue text, the source code text) and further data such as developer-specific information. In their final experiment, they used the trained machine learning classifiers to identify the matching issues for commits without issues and achieved an average recall of 91.6% and precision of 17.3%. A direct comparison with *IR*-based link creation is missing. However, since these results are quite similar to what others have achieved by relying on *IR* and *ITS* data alone [Merten et al., 2016b], it seems that the usage of *IR* to train machine learning classifiers results in the same low precision values as when relying on *IR* alone.

2.3 Measurement Fundamentals

To rate the quality of trace link creation approaches objectively, a link set of the real trace links and measures in order to rate the link candidates created by an approach are necessary. Thus, in the following Section 2.3.1 the term gold standard, which refers to the set of real trace links used in trace link approach evaluations, is introduced. In Section 2.3.2, the evaluation measures used to rate the quality of trace link creation approaches by comparing created link candidates with links from an gold standard are introduced. This also includes typical categories for the achieved quality of an trace link creation approach.

2.3.1 Gold Standard

To evaluate the quality of trace link creation approaches, a gold standard which consists of the set of all correct trace links for a given set of artefacts is important [Borg et al., 2014, Gotel et al., 2012b]. To create such a gold standard, it is necessary to manually check whether trace links exist for each pair of artefacts.

The creation of such a gold standard is labor-intensive, especially for large realistic real world datasets. Moreover, experts with knowledge about all involved artefacts are required for the creation of a gold standard. The lack of experts with that knowledge, i.e. a person who has the required knowledge, can also be a reason why the creation of a complete gold standard in large projects is simply not possible [Borg et al., 2014].

Therefore many trace link creation approaches use datasets which are specifically created for the purpose of evaluation, e.g. within a student project [De Lucia et al., 2007]. This is also one of the reasons why two of the performed evaluation studies of this thesis used student projects. Thus it was possible to create the gold standard in parallel to the projects.

2.3.2 Evaluation Measures

Trace links which are initially created by a trace link creation approach are often called trace link candidates, and only after the trace link candidates have been vetted by a human analyst they are called trace links. Also, some trace link creation approaches consist of multiple steps for the creation of trace links, e.g. when in a first step *IR* is used to create an initial set of link candidates and in a second step further information such as *SCS* is used to improve the initial set of link candidates to the final set of trace links.

This also applies to the *ILog* approach of this thesis. *ILog* consists of multiple steps and the links resulting from the intermediate steps are called trace link candidates, while the links from the final step are called trace links. Furthermore, in studies which evaluate trace link creation approaches, by comparing links created by an approach to links of a gold standard, the links from the approach are called link candidates and the links from the gold standard links [Borg et al., 2014].

Based on the link set from a gold standard and the link candidate set which is created by a trace link creation approach, the evaluation measures introduced in the following are used to rate the quality of a trace link creation approach, i.e. the final created trace links. Precision (P) is the amount of correct links, i.e. *True Positive* (*TP*), within all links found by an approach. The latter (i.e all links found by an approach) is the sum of *TP* and incorrect links, i.e. *False Positive* (*FP*). Recall (R) is the amount of *TP* links found by an approach within all existing correct links. The latter (i.e. all existing correct links) is the sum of *TP* and *False Negative* (*FN*), i.e. not found, links:

$$P = \frac{TP}{TP + FP} \qquad R = \frac{TP}{TP + FN} \qquad F_\beta = (1 + \beta^2) \cdot \frac{P \cdot R}{(\beta^2 \cdot P) + R}$$

F_β-scores combine the results for P and R in a single measurement to judge the accuracy of a trace link creation approach. As shown in the equation for F_β above,

β can be used to weight P in favor of R and vice versa. Because the research goal of this thesis is to directly provide automatically created trace links to developers, the focus for *ILog's* evaluation is to emphasize P, but still consider R. Therefore, the evaluation studies of this thesis use $F_{0.5}$ which weights P twice as much as R. In addition, in the evaluation studies F_1-scores are also calculated to compare the results with those of others.

Relative recall is used if it is not possible to obtain all correct values for a dataset due to the size of the dataset [Frické, 1998]. It is a well-established standard measure in the domain of web search engine performance and quality measuring where the missing of a complete gold standard is common [Kumar and Prakash, 2009]. Relative recall uses all correct links available as a comparison measure to calculate the recall of a single approach. The calculation of relative recall is defined as:

$$R_n^r = \frac{TP_n}{\sum_{i=1}^{i} TP_i} \qquad R_{IR}^r = \frac{TP_{IR}}{TP_{IR} + TP_{ILog}} \qquad R_{ILog}^r = \frac{TP_{ILog}}{TP_{ILog} + TP_{IR}}$$

The first equation shows the general case in which the relative recall R_n^r for the approach n is calculated on the basis of the fraction of all correct links (TP_n) of the approach n and the sum of all correct links from all approaches which have been used for link creation ($\sum_{i=1}^{i} TP_i$). The further equations show the cases in which *IR* and *ILog* have been used for link creation and the respective relative recall R_{IR}^r for *IR* and R_{IL}^r for *ILog*. The F_β-score calculation with relative recall is identical to real recall as it was previously introduced. In this thesis, relative recall has been used in one evaluation study (cf. Chapter 8) in which interaction data from an open source project was used. Due to the project's size according to involved requirements and the source code files, it was not possible to create a complete gold standard.

According to Hayes et al. [2006], the values for P and R of *IR*-based trace link creation approaches for structured requirements can be categorized according to three quality levels. *Acceptable* values for R are between 60 and 69% and for P between 20 and 29%. *Good* values for R are between 70 and 79% and for P between 30 and 49% and *excellent* values for R are between 80 and 100% and for P between 50 and 100%. Merten et al. [2016b] reported varying results for using *IR* on unstructured requirements data from *ITS*, in which they tried to achieve 100% for R with different *IR* techniques and different preprocessing steps. According to their study their best values for P were up to 11%. In considering other approaches for link creation between code and requirements using open source projects as data sources, Ali et al. [2013] used *VSM* for trace link creation and achieved similar but also very project-specific results for P (between 15% and 77%). De Lucia et al. [2007] report values of 90% for R and 25% for P for link creation between structured requirements and source code by using *LSI* in combination with categorization. The reported categorize and values for P and R are used to rate the results of the performed evaluation studies.

Part II

Problem Investigation

Chapter 3

Quality of Trace Link Creation: State of the Art

This chapter presents a review of existing *IR*-based trace link creation approaches. In the design cycle of the thesis, the review is the core part of the task of problem investigation. It answers the general research question: *Which automatic trace link creation approaches exist and what is the quality of the resulting links?* The results of the review show the insufficiency of current automatic trace link creation approaches to create trace links for continuous and direct usage. That is to say, the precision of these approaches is insufficient and the link candidates which result from them need further manual assessment.

Section 3.1 introduces the method concerning how the trace link creation approach review has been performed, including the detailed research questions and the rationale for them. Section 3.2 presents the results by answering the previously stated research question, and includes a concluding discussion which further motivates the creation of the *ILog* Approach.

3.1 Method

Instead of performing a *SLR*, this trace link creation approach review is based on three existing reviews about trace link creation. However, expect for the part concerning publication identification, the review follows the steps of a *SLR*, as suggested in the *SLR* guidelines of Kitchenham and Charters [2007]. Section 3.1.1 introduces the method of the review. Section 3.1.2 states the research questions and attributes that are necessary to answer these research questions. Section 3.1.3 introduces the three literature review publications about trace link creation approaches which were used and presents the way that the trace link creation approaches reviewed in this chapter have been selected.

3.1.1 Review Method

According to the *SLR* guidelines of Kitchenham and Charters, after defining the research questions and setting up a search strategy, it is necessary to define the exclusion and inclusion criteria for the literature which has been reviewed, and to create a classification schema which allows for a systematic answer of the stated research questions.

By using the existing literature reviews, basic exclusion criteria to ensure the quality of the reviewed approaches have already been applied, e.g. English language, peer reviewed, etc. The primary inclusion criterion was that a publication must describe a trace link creation approach in which trace links between source code and other artefacts are created. The trace link creation approaches have been limited to include the source code as source/ target for trace links, since the *ILog* approach is intended for link creation between requirements and source code. The secondary inclusion criterion was that an evaluation of the approach was performed and is described in the publication with results that enable a quality rating of the approach regarding continuous and direct link usage.

The approach classification schema used in this review is defined by the attributes which have been used for answering the research questions.

3.1.2 Research Questions

Table 3.1. Trace Link Creation Review Research Questions and Attributes

	Research Question	Attributes
RQ1	What are the characteristics of automatic trace link creation approaches?	Standardized description of the automatic trace link creation approach
RQ1.1	Which automatic trace link creation approaches exist?	Publication describing the automatic trace link creation approach
RQ1.2	Which artefacts are linked with each other?	Source and target artefacts of trace links
RQ1.3	What is the intended link creation time and frequency?	Link creation time *(retrospective, during project)* and frequency *(continuous, manually triggered)*
RQ1.4	What are the means of link creation used?	The link creation method used *(IR-technique, initial (from scratch) creation of links, use of trace link improvement techniques)*
RQ1.5	What is the quality of the created trace links?	Trace link quality *(precision, recall)*

Table 3.1 shows a refinement of the general research question: *Which automatic trace link creation approaches exist and what is the quality of the resulting links?* The table also includes the attributes necessary to answer the research questions.

RQ1 asks for the characteristics of trace link creation approaches. In *RQ1.1* (existing approaches) the publication for each trace link creation approach is identified. The subsequent RQs – from *RQ1.2* to *RQ1.5* – collect the data required a standardized description of the approaches. *RQ1.2* shows which artefacts are linked with source code files. *RQ1.3* shows the intended link creation time, which makes

it possible to judge if an approach is suitable for continuous link creation. *RQ1.4* shows the details about the *IR*-technique used and states whether and what kind of link improvement techniques have been applied. *RQ1.5* shows the resulting link quality of an approach, which makes it possible to judge if the resulting links of an approach are suitable for direct use.

3.1.3 Overview of Literature Selection

For the review of automatic trace link creation approaches, three existing literature reviews in the field of traceability research were selected[1]. The selection criterion for the reviews was to address the current state of the art in traceability research of automatic trace link creation. In the following, the key characteristics of the three reviews are summarized.

In TR1, Gotel et al. [2012b] align the traceability approaches presented with the grand challenges in traceability research [Cleland-Huang et al., 2014]. In TR2, Borg et al. [2014] focus on approaches using *IR*-based traceability methods. In TR3, Cleland-Huang et al. [2014] present an updated report of the grand challenges in traceability research, including ongoing trends.

Those traceability approaches which were discussed in these three review studies, and which satisfy the inclusion criteria to create trace links between requirements and source code (I1) and to be assessable according to continuous link creation and direct link usage (I2), were selected.

3.2 Results

In Section 3.2.1 the research questions are answered by presenting the publications for the trace link creation approaches that were found and a standard description of the approaches, using the attributes of the research questions. In Section 3.2.2 the results are summarized and discussed regarding the research goals of the thesis and implications for the design of the *ILog* approach.

3.2.1 Overview of Trace Link Creation Approaches and Answers to the Research Questions

RQ1.1: Which Automatic Trace Link Creation Approaches Exist

Table 3.2 answers *RQ1.1* (which approaches exist) by presenting the overview of the 12 identified trace link creation approaches and the publication for each approach.

[1]Initially the review study from Nair et al. [2013], which concerns traceability approaches presented at the *International Requirements Engineering Conference*, was also selected to identify automatic trace link creation approaches. However, after the consolidation of all selected trace link creation approaches from all review studies, it turned out that only older publications for approaches, which were already identified in the other review studies, were present in [Nair et al., 2013]. Thus only the most current publications as found in the other review studies were used.

Table 3.2. Primary Publications for the Identified Trace Link Creation Approaches

ID	Trace Link Creation Approach Publication
T01	Yoëlle S. Maarek, Daniel M. Berry, and Gail E. Kaiser. An Information Retrieval Approach For Automatically Constructing Software Libraries. *IEEE Transactions on Software Engineering (TSE)*, 17(8):800–813, 1991
T02	Giuliano Antoniol, Gerardo Canfora, Gerardo Casazza, Andrea De Lucia, and Ettore Merlo. Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering (TSE)*, 28(10): 970–983, 2002
T03	Andrian Marcus and Jonathan I. Maletic. Recovering Documentation-to-Source-Code Traceability Links using Latent Semantic Indexing. In *Proceedings of the 25th International Conference on Software Engineering (ICSE)*, pages 125–135, Portland, OR, USA, 2003. ACM/IEEE
T04	Andrea De Lucia, Fausto Fasano, Rocco Oliveto, and Genoveffa Tortora. Enhancing an Artefact Management System with Traceability Recovery Features. In *Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM)*, pages 306–315, Chicago, IL, USA, 2004. IEEE
T05	Jane Cleland-Huang, Raffaella Settimi, Oussama BenKhadra, Eugenia Berezhanskaya, and Selvia Christina. Goal-centric Traceability for Managing Non-functional Requirements. In *Proceedings of the 27th International Conference on Software Engineering (ICSE)*, pages 362–371, St. Louis, MO, USA, 2005. ACM/IEEE
T06	Leonardo Gresta Paulino Murta, André van der Hoek, and Cláudia Maria Lima Werner. ArchTrace: Policy-Based Support for Managing Evolving Architecture-to-Implementation Traceability Links. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 135–144, Tokyo, Japan, 2006. IEEE
T07	Xuchang Zou, Raffaella Settimi, and Jane Cleland-Huang. Phrasing in Dynamic Requirements Trace Retrieval. In *Proceedings of the 30th Annual International Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 265–272, Chicago, IL, USA, 2006. IEEE
T08	Andrea De Lucia, Fausto Fasano, Rocco Oliveto, and Genoveffa Tortora. Recovering Traceability Links in Software Artifact Management Systems using Information Retrieval Methods. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 16(4):1–50, 2007
T09	Malcom Gethers, Rocco Oliveto, Denys Poshyvanyk, and Andrea De Lucia. On Integrating Orthogonal Information Retrieval Methods to Improve Traceability Recovery. In *Proceedings of the 27th IEEE International Conference on Software Maintenance (ICSM)*, pages 133–142, Williamsburg, VA, USA, 2011. IEEE
T10	Achraf Ghabi and Alexander Egyed. Code Patterns for Automatically Validating Requirements-to-Code Traces. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 200–209, Essen, Germany, 2012. ACM
T11	Alexander Delater and Barbara Paech. Analyzing the Tracing of Requirements and Source Code during Software Development. In *Proceedings of the 19th International Working Conference - Requirements Engineering: Foundation for Software Quality (REFSQ)*, volume 7830 of *Lecture Notes in Computer Science (LNCS)*, pages 308–314, Essen, Germany, 2013. Springer
T12	Bogdan Dit, Meghan Revelle, and Denys Poshyvanyk. Integrating information retrieval, execution and link analysis algorithms to improve feature location in software. *Empirical Software Engineering*, 18(2):277–309, 2013

Table 3.3 shows in which of the three review studies the approaches were found. Three of the approaches were include in two review studies. Review study TR1 contains five, TR2 contains three and TR3 contains seven approaches, respectively.

In the following, each of the 12 trace link creation approaches is summarized based on the characteristics shown in Table 3.4.

In T01, Maarek et al. [1991] describe an automatic trace link creation approach that creates links between requirements and source code. The link creation is intended for a retrospective application, and uses the *IR* technique of *lexical affinities*[2] for an initial automatic link creation. The best achieved precision is 52% and the best achieved recall is 90%.

In T02, Antoniol et al. [2002] describe an automatic trace link creation approach that creates links between requirements and source code. The link creation

[2]A *lecial affinities* is pair of co-occurring terms with a certain statistical relevance. In the trace link creation of the described approach *lexical affinities* are used to judge if artefacts should be linked with each other.

Table 3.3. Sources of Trace Link Creation Approaches

Review	T01	T02	T03	T04	T05	T06	T07	T08	T09	T10	T11	T12	∑
TR1	✓		✓		✓		✓	✓					5
TR2		✓					✓	✓					3
TR3	✓			✓		✓			✓	✓	✓	✓	7

Table 3.4. Properties of Trace Link Creation Approaches

Approach properties			T01	T02	T03	T04	T05	T06	T07	T08	T09	T10	T11	T12	∑
Links Code with		Functional Requirements	✓	✓	✓	✓			✓	✓		✓	✓	✓	9
		Use Cases				✓				✓					2
		Features												✓	1
		Non-functional req.					✓								1
		Design models			✓					✓	✓				3
		Architecture models						✓							1
		Test cases				✓				✓					2
Link Creation	Time	Retrospective	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓	11
		During dev.											✓		1
	Intention	Initial	✓	✓	✓	✓	✓		✓	✓	✓		✓	✓	10
		Improvement					✓	✓	✓	✓		✓			5
		Manual effort					✓	✓							2
	Technique	*IR*	✓	✓	✓	✓	✓		✓	✓				✓	8
		Lexical affinities	✓												1
		VSM		✓										✓	2
		LSI			✓	✓	✓		✓	✓				✓	6
		Probab. model					✓		✓						2
		NLP						✓	✓		✓				3
		(*POS*) Phrasing							✓						1
		Topic modelling									✓				1
		Rules						✓				✓	✓		3
		SCS										✓			1
		(Com.) Revision											✓		1
Link Quality	Precision		52	17	43	14	51	95	39	25	38	95	88	-	
	Recall		90	100	71	100	87	89	90	90	-	96	93	-	

is intended for a retrospective application, and uses *IR* with *VSM* for an initial automatic link creation. The best achieved precision is 17% and the best achieved recall is 100%.

In T03, Marcus and Maletic [2003] describe an automatic trace link creation approach that creates links between requirements, design models and source code. The link creation is intended for a retrospective application, and uses *IR* with *LSI* for an initial automatic link creation. The best achieved precision is 43% and the best achieved recall is 71%.

In T04, Lucia et al. [2004] describe an automatic trace link creation approach that creates links between the requirements specified as use cases, test cases and source code. The link creation is intended for a retrospective application, and uses *IR* with *LSI* for an initial automatic link creation. The best achieved precision is 14% and the best achieved recall is 100%.

In T05, Cleland-Huang et al. [2005] describe an automatic trace link creation approach that creates links between non-functional requirements and source code.

The link creation is intended for a retrospective application, and it uses *IR* with *LSI* and a probabilistic model for automatic trace link creation. The probabilistic model used is intended to improve the resulting links in comparison to using only *LSI*. However, it also requires a manual vetting of the link candidates. The best achieved precision is 51% and the best achieved recall is 87%.

In T06, Murta et al. [2006] describe an automatic trace link creation approach that improves existing links between architecture models and source code. The link improvement is intended for a retrospective application, and uses *NLP*-based rules for the semi-automatic (i.e. it also requires manual effort) improvement of the previously created links. The best achieved precision is 95% and the best achieved recall is 89%.

In T07, Zou et al. [2006] describe an automatic trace link creation approach that creates links between requirements and source code. The link creation is intended for a retrospective application, and it improves upon the T05 probabilistic model-based approach by additionally using the *NLP* technique of *Part of Speech Tagging* (*POS*) and phrasing. The best achieved precision is 63% and the best achieved recall is 90%.

In T08, De Lucia et al. [2007] describe an automatic trace link creation approach that creates links between requirements in the form of use cases, test cases, design models and source code. The link creation is intended for a retrospective application, and uses *IR* with *LSI* for initial automatic creation of trace links. Furthermore, the approach improves links by applying categorisation, i.e. by using the type of linked artefacts as an additional measure for link creation. The best achieved precision is 33% and the best achieved recall is 95%.

In T09, Gethers et al. [2011] describe an automatic trace link creation approach that creates links between design models and source code. The link creation is intended for a retrospective application, and it uses the *NLP* technique topic modelling for initial automatic link creation. The best achieved precision is 38% but achieved recall values are not reported.

In T10, Ghabi and Egyed [2012] describe an automatic trace link creation approach that improves links between requirements and source code. The link creation is intended for a retrospective application, and uses rules based on *SCS* for the automatic improvement of previously created links. The best achieved precision is 95% and the best achieved recall is 96%.

In T11, Delater and Paech [2013] describe a semi-automatic trace link creation approach that creates links between requirements and source code. The link creation is intended for application during the development, and it uses rules based on recorded interactions during the implementation of requirements and manually created relations between commits and issues for an initial automatic link creation. The best achieved precision is 88% and the best achieved recall is 93%.

In T12, Dit et al. [2013] describe an automatic trace link creation approach that creates links between requirements which are specified as features and source code. The link creation is intended for a retrospective application, and uses *IR* with *VSM* and *LSI* for an initial automatic link creation. Instead of precision and recall, the performed study evaluated the effectiveness improvements of the approach presented (combination of multiple *IR* techniques) in comparison to standard approaches (single *IR* technique). The approach achieved an effectiveness improvement of 87%.

RQ1.2: Linked Artefacts and RQ1.3: Link Creation Time

The first rows of Table 3.4 show which artefacts are linked with source code files in the approaches. Most of the approaches link with requirements (9 functional, 1 non-functional). Other artefacts which are often linked are model elements from design (3) or architectural (1) models. Three approaches created links from the source code to two different kind of artefacts. For three approaches linking to requirements, the requirement format is explicitly defined as the use cases T04 and T08 or as feature T12.

For all but one (11) of the approaches, the intended link creation time is retrospectively and manually triggered. Only the approach T11 is designed for continuous link creation during a project but requires manual assignment of issues to commits for link creation.

RQ1.4: Means of Link Creation

The rows in the *Link Creation* segment of Table 3.4 characterise the means of link creation used in the approaches. 10 of the approaches are intended for the initial creation of links which utilize the contents of linked artefacts. Five of the approaches apply some kind of improvement technique, out of which three do this in conjunction with the initial link creation, e.g. by not only using the results of an *IR* technique but also by considering different data, such as the artefact types. The other two approaches which apply improvement techniques do not directly specify how the initial link creation is performed. In addition, two of the approaches which also apply improvement techniques require a manual processing of the resulting link candidates in order to achieve the reported precision and recall values.

Eight of the approaches use the *IR* techniques, lexical affinities (1), *VSM* (2), *LSI* (6), or probabilistic models (2). The two approaches which use probabilistic models use these models as an improvement for only applying *LSI*.

Three of the approaches use *NLP*, of which one is for improving *IR*-based link creation, while the other two are used as a primary link creation technique. The *NLP* techniques used are phrasing together with *POS* and topic modelling.

Three of the approaches use rules for link creation. One of them does so together with *NLP*, one with *SCS*, and one with revisions in a *VCS* created by commits.

RQ1.5: Link Quality

The rows in the *Link Quality* segment at the bottom of Table 3.4 summarize the findings regarding link creation quality in terms of their respective precision and recall. The performance of the investigated approaches is uneven. Whereas retrospective approaches for initial link creation reach a precision between 14% and 52%, approaches with link improvement and the prospective approach T11 achieve a precision between 33% and 95%, respectively. The achieved recall values are between 71% and 100%. However, the two approaches which achieve 100% recall also have the lowest precision values of 17% and 13%, respectively.

3.2.2 Summary and Discussion

Almost all approaches create links retrospectively by analysing the contents of artefacts. First, retrospective approaches aim at creating an initial set of trace links between source code files and the target artefacts by applying an *IR* or *NLP* technique or a rule-based link inference mechanisms. Second, they focus on further improving precision and recall by extending the initial creation techniques with other further techniques, or by applying further techniques after the initial creation and by asking developers to select the appropriate links. Only T11 supports a semi-automatic creation of trace links during implementation by tracking which requirements developers view while writing code. Overall, beside the approach T11, none of the other approaches directly provides the links after their creation. The link quality of retrospective approaches which create an initial set of trace links is insufficient for direct use, due to their low precision. High precision and recall can only be achieved together based on manual improvement activities or interaction tracking of developers.

None of the reviewed approaches fully accomplishes the research goal of this thesis, which is to continuously provide directly usable trace links during a project without requiring additional manual effort from developers. Either the link quality of the approaches is insufficient, or manual effort is required to achieve good precision, or approaches assume that links already exist and only an improvement of existing links is performed. However, some of the aspects of the reviewed approaches are considered for the design of the *ILog* approach. These aspects include the usage of interaction tracking and recording for link creation, the use of data from commits to a *VCS*, the principle of using a further technique to improve links after their initial creation, and the utilization of *SCS* as an improvement technique.

Chapter 4

Trace Link Maintenance: State of the Art

As a second problem which is investigated in the thesis, this chapter provides a *Systematic Literature Review* (*SLR*) concerning the maintenance of trace links (*TM*). Links have to be maintained along with the changes of linked artefacts during the progress of a project [Wohlrab et al., 2016, Maro et al., 2016]. They can become obsolete due to changes in the linked artefacts and, when they do, they become useless. *TM* is a crucial part of the complete trace link management process [Cleland-Huang et al., 2014]. Therefore, this *SLR* collects the state of the art of *TM*.

Another reason to create this overview is to identify approaches suitable for the maintenance extension of the *ILog* approach which is developed in this thesis. The following Section 4.1 describes the research method. Section 4.2 describes the process of publication search and selection, and Section 4.3 provides the results.

4.1 Method

The *SLR* follows the guidelines of Kitchenham and Charters [2007]. Section 4.1.1 first introduces the review method which is used. Section 4.1.2 states the research questions and the rationale for them.

4.1.1 Review Method

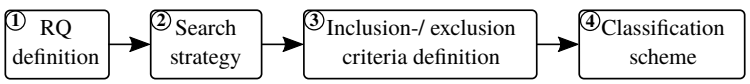

Figure 4.1. Systematic Literature Review Method Approach

The *SLR* follows the approach of Kitchenham and Charters [2007]. As shown in Figure 4.1, the first step is to define the research questions. The second step is to set

up a search strategy. Kitchenham and Charters suggests to use online databases to initially identify relevant publications. To query an online database, a search string consisting of search terms which reflect the research questions and the attributes that are required to answer the research questions is necessary. Due to the different search interfaces of the online databases (i.e. use of logical expressions, restrictions to certain files and meta data attributes, etc.), an online database-specific adaptation of the search query and the usage of search interface-specific filtering options are necessary. The searching of online databases can be supplemented with manual target search.

Furthermore, inclusion and exclusion criteria for publications which are to be found by the keyword search have to be defined. Kitchenham and Charters define the general criteria for this as the following: a publication has to be available in English full text, it has to be peer reviewed, and there should be empirical results in the publication.

These general criteria are then supplemented by criteria which are specific to the research questions and the attributes that are required to answer the research questions. For this *SLR*, the research question-specific inclusion criteria are that the publications need to describe a *TM* approach.

Finally, a classification scheme for the identified publications is needed. This is necessary to compare the different approaches. In addition, it is also helpful to extract general publication data and meta-data, such as the names of the authors, year, title, venue.

Table 4.1. Trace Link Maintenance SLR Research Questions and Attributes

	Research Question	**Attributes**
RQ1	What are the characteristics of *TM* approaches?	Standardized description of the *TM* approach
RQ1.1	Which *TM* approaches exist?	Publication describing trace link maintenance approach
RQ1.2	Which artefacts are linked with each other?	Source and target artefacts of trace links
RQ1.3	Which other data sources/ artefacts are used for *TM*?	Used data sources/ artefacts
RQ1.4	How is *TM* performed and how are the artefacts and the trace links used for that?	Description of *TM* approach by a generic 4 step process
RQ1.5	To what extent and how are *TM* approaches automated?	Which part is automated? *(detection, execution)* Degree of automation *(automatic, semi-automatic, manual, indicated by manual/ monitoring for the detection steps and tool-based/ manual in the determination & execution steps)*
RQ1.6	How are the approaches evaluated and what are the evaluation results?	Type of Evaluation *(qualitative, quantitative)* Evaluation results *(precision, recall and f-measures if a quantitative evaluation has been performed)*

4.1.2 Research Questions

The general research questions of this *SLR* is: *What are the characteristics of* TM *approaches?* Table 4.1 shows a refinement of the general research question and the attributes of the approaches that are necessary to answer the research questions.

RQ1 enquires into the characteristics of *TM* approaches. In *RQ1.1* (existing approaches) the primary publication for each *TM* approach is identified. The subsequent *RQs* – from *RQ1.2* to *RQ1.5* – collect the data required for a standardized description of a *TM* approach. *RQ1.2* (linked artefacts) helps to distinguish the range of use for a *TM* approach. With the answers from *RQ1.3* (data) and *RQ1.4* (techniques, algorithms, etc.) it is possible to determine what is necessary for applying a *TM* approach. *RQ1.5* helps to understand which parts of a *TM* approach are automated. This is also interesting in connection with the integration of *TM* in the *ILog* approach, since one of the research goals for *ILog* is to add as little additional manual effort as possible. Finally, *RQ1.6* helps to rate the maturity of a *TM* approach, which is also important when considering the integration in *ILog*.

4.2 Publication Search

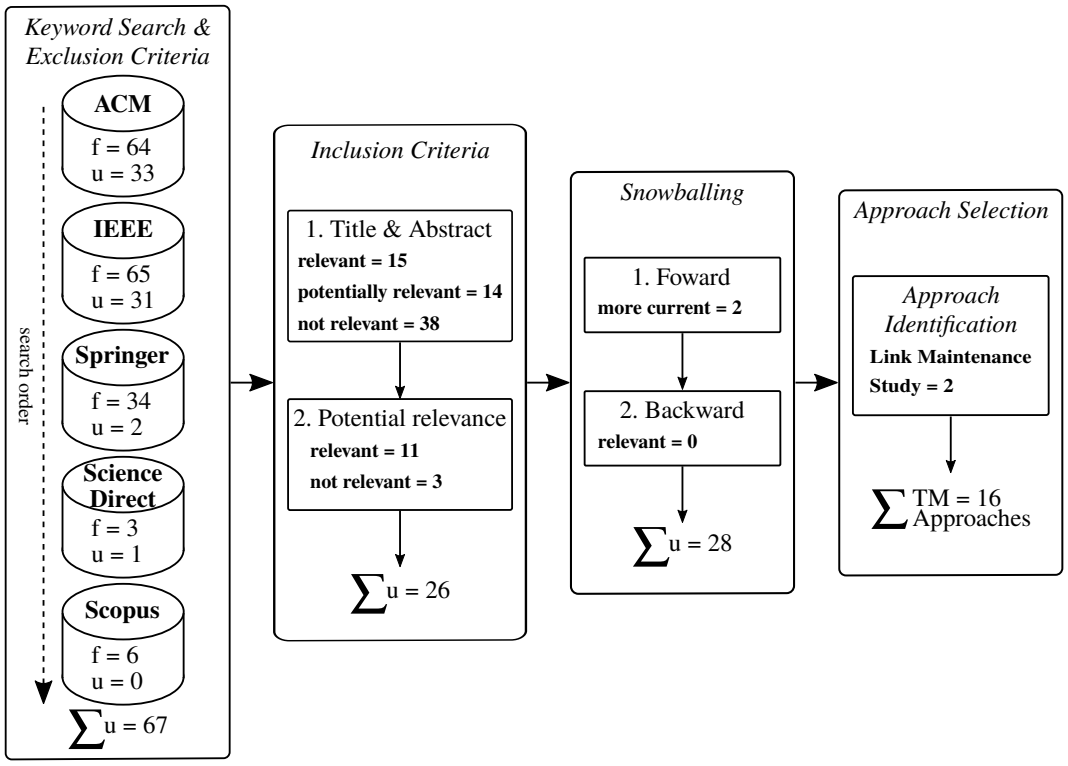

Figure 4.2. Publication Search and Trace Link Maintenance Approach Identification

Figure 4.2 shows the overview of the publication search and the selection of *TM* approaches from the identified publications. The steps of the publication search and the identification of *TM* approaches are described in the following section.

Listing 4.1. Keyword Search Query

```
"traceability maintenance" OR "trace link maintenance"
```

Listing 4.1 shows the keyword query used for the publication search. The query was tested with a pre-search by using *Google Scholar*[1]. Due to the amount and type of publications found in the pre-search, the search terms have been retained.

As shown in Figure 4.2, five scientific online databases have been used to search for publications with the keywords from above in the order shown. For some of the databases used, the keyword query of Listing 4.1 was adapted due to technical concerns about the database's search engines; for example, for some of the databases it was necessary to explicitly select the attributes of publications (such as title, abstract, content, etc.) to be used when searching.[2] The numbers for each database refer to the publications found (f) and used (u). If publications were found in multiple databases, only their first occurrence is counted. This resulted in 67 distinct publications obtained from all scientific online databases.

Table 4.2. Exclusion (E_n) and Inclusion (I_n) Criteria for TM Publications

Criteria	Description
E_1	publication is published before 2000
E_2	publication is not written in English
E_3	publication is not peer reviewed
I_1	publication describes a trace link maintenance approach or a trace link creation approach which includes trace link maintenance

To filter the identified publications, the exclusion and inclusion criteria shown in Table 4.2 have been applied. The exclusion criteria E_1 to E_3, which aimed to ensure quality and timeliness, have been applied within the filtering functionality of the databases' search interfaces. Thus, the number of publications used which are shown in the first step of Figure 4.2 already comprise the application of E_1 to E_3.

The inclusion criterion I_1 was applied in a second step after the overall number of 67 publications was obtained. This criterion was first only applied to the publication title and abstract. 15 publications were identified as relevant, 14 publications as potentially relevant, and 38 as not relevant. After this, for the 14 potentially relevant publications the complete publication text was used to decide on the question of their inclusion. This resulted in a further 11 relevant and 3 not relevant publications. Thus, finally 26 relevant publications were identified after the application of the inclusion criterion.

[1] `https://scholar.google.com/`, *Google Scholar* publication search

[2] More details about the *TM SLR* can be found in the supplementary material Section A of the thesis's appendix. Within this, details about the performed online databases specific query adoption are to be found in Section A.1.2

To ensure that all relevant publications and the latest publication for a *TM* approach were both obtained, forward and backward snowballing was performed. By forward snowballing, two more current publications for two of the identified approaches were found, but no publications were found about approaches which have now already been covered. Backward snowballing did not bring any new insights. This resulted finally in 28 relevant publications after snowballing.

The 28 relevant publications contained 16 distinct *TM* approaches, of which four of the 16 approaches were described in multiple publications. Two publications described studies about *TM*, but did not contain their own *TM* approach. For the 16 *TM* approaches, one primary publication was chosen due to the novelty of the publication, the comprehensiveness of the *TM* approach description, and the evaluation performed.[3]

4.3 Results

This section presents the results of the *SLR*. First, a generic trace link maintenance process is introduced, which is used in Section 4.3.1 to answer the research questions and in Section 4.3.2 to discuss the results.

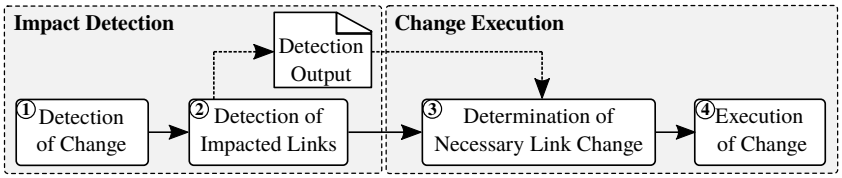

Figure 4.3. Trace Link Maintenance Process

As shown in Figure 4.3, the *TM* approaches are characterized on the basis of a generic process. This process consists of four steps that are separated in an impact detection and execution part summarized in the following:

Impact Detection consists of the detection of a change in linked artefacts and the subsequent detection of impacted links. The resulting output contains the impacted artefacts and links and potential further data.

Change Execution consists of the determination of necessary link changes based on the previously generated output and the execution of those changes.

In this *SLR* it is assumed that trace links are maintained along with the maintenance of all other artefacts in the progress of a project. Thus, an artefact change triggers the maintenance of trace links. However, there are *TM* approaches which only analyse the current state of the project artefacts and then both rate existing links

[3]Details about selection of a different publication for approaches with multiple publications can be found in Section A.1.3 of the appendix.

and find missing links. In these cases, the authors do not propose the application of their approaches for every project change. This kind of approaches are threatened as *TM* approach by adding a manual change detection step.

4.3.1 Overview of Trace Link Maintenance Approaches and Answers to the Research Questions

Table 4.3. Primary Publications for Identified TM Approaches

ID	*TM* Approach Primary Publication
P01	Ove Armbrust, Alexis Ocampo, Jurgen Munch, Masafumi Katahira, Yumi Koishi, and Yuko Miyamoto. Establishing and Maintaining Traceability Between Large Aerospace Process Standards. In *Proceedings of the ICSE Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE@ICSE)*, pages 36–40, Vancouver, BC, Canada, 2009. IEEE
P02	Dominique Blouin, Matthias Barkowski, Melanie Schneider, Holger Giese, Johannes Dyck, Etienne Borde, Dalila Tamzalit, and Joost Noppen. A Semi-Automated Approach for the Co-Refinement of Requirements and Architecture Models. In *Proceedings of the 25th IEEE International Requirements Engineering Conference Workshops (REW)*, pages 36–45, Lisbon, Portugal, 2017. IEEE
P03	Jane Cleland-Huang, Carl K. Chang, and Mark Christensen. Event-based traceability for managing evolutionary change. *IEEE Transactions on Software Engineering (TSE)*, 29(9):796–810, 2003
P04	Jane Cleland-Huang, Patrick Mäder, Mehdi Mirakhorli, and Sorawit Amornborvornwong. Breaking the Big-Bang Practice of Traceability: Pushing Timely Trace Recommendations to Project Stakeholders. In *Proceedings of the 20th IEEE International Requirements Engineering Conference (RE)*, pages 231–240, Chicago, IL, USA, 2012. IEEE
P05	Nikolaos Drivalos-Matragkas, Dimitrios S. Kolovos, Richard F. Paige, and Kiran J. Fernandes. A State-based Approach to Traceability Maintenance. In *Proceedings of the 6th ECMFA Traceability Workshop (ECMFA-TW)*, ECMFA-TW '10, pages 23–30, Paris, France, 2010. ACM
P06	Markus Fockel, Jörg Holtmann, and Jan Meyer. Semi-automatic Establishment and Maintenance of Valid Traceability in Automotive Development Processes. In *Proceedings of the 2nd International Workshop Software Engineering for Embedded Systems (SEES)*, pages 37–43, Zurich, Switzerland, 2012. IEEE
P07	Vincenzo Gervasi and Didar Zowghi. Supporting Traceability Through Affinity Mining. In *Proceedings of the 22nd IEEE International Requirements Engineering Conference (RE)*, pages 143–152, Karlskrona, Sweden, 2014. IEEE
P08	Achraf Ghabi and Alexander Egyed. Code Patterns for Automatically Validating Requirements-to-Code Traces. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 200–209, Essen, Germany, 2012. ACM
P09	Markus Kleffmann, Matthias Book, and Volker Gruhn. Towards Recovering and Maintaining Trace Links for Model Sketches Across Interactive Displays. In *Proceedings of the 7th International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE@ICSE)*, pages 23–29, San Francisco, CA, USA, 2013. IEEE
P10	Patrick Mäder and Orlena Gotel. Towards Automated Traceability Maintenance. *Journal of Systems and Software*, 85(10):2205–2227, 2012a
P11	Salome Maro and Jan-Philipp Steghöfer. Capra: A Configurable and Extendable Traceability Management Tool. In *Proceedings of the 24th IEEE International Requirements Engineering Conference (RE)*, pages 407–408, Beijing, China, 2016. IEEE
P12	Richard F. Paige, Nikolaos Drivalos, Dimitrios S. Kolovos, Kiran J. Fernandes, Christopher Power, Goran K. Olsen, and Steffen Zschaler. Rigorous identification and encoding of trace-links in model-driven engineering. *Software and Systems Modeling*, 10(4):469, 2011
P13	Mona Rahimi, William Goss, and Jane Cleland-Huang. Evolving Requirements-to-Code Trace Links across Versions of a Software System. In *Proceedings of the 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 99–109, Raleigh, NC, USA, 2016. IEEE
P14	Hannes Schwarz, Jürgen Ebert, and Andreas Winter. Graph-based traceability: a comprehensive approach. *Software and Systems Modeling*, 9(4):473–492, 2010
P15	Andreas Seibel, Regina Hebig, and Holger Giese. Traceability in Model-Driven Engineering: Efficient and Scalable Traceability Maintenance. In *Software and Systems Traceability*, pages 215–240. Springer, 2012
P16	Pan Ying, Tang Yong, and Ye Xiaoping. Software Artifacts Management Based on Dataspace. In *Proceedings of the WASE International Conference on Information Engineering*, volume 2 of *ICIE '09*, pages 214–217, Taiyuan, Chanxi, China, 2009. IEEE

RQ1.1: Which TM Approaches Exist?

Table 4.3 answers *RQ1.1* (which *TM* approaches exist?) by presenting the overview of the 16 identified *TM* approaches and the primary publication for each approach.

Table 4.4. Trace Link Source and Target Artefacts

Linked Artefacts	P01	P02	P03	P04	P05	P06	P07	P08	P09	P10	P11	P12	P13	P14	P15	P16	∑
Software artefacts					✓									✓		✓	3
Development standard	✓																1
Source code			✓					✓			✓		✓			✓	5
Requirements		✓	✓	✓		✓	✓	✓		✓			✓				8
(UML) Model element		✓		✓	✓	✓				✓	✓	✓		✓	✓		9
Sketch									✓								1

Table 4.5. Trace Link Maintenance Process

	Action	P01	P02	P03	P04	P05	P06	P07	P08	P09	P10	P11	P12	P13	P14	P15	P16	∑
S1	Manual Indication of changed artefacts	✓							✓				✓	✓		✓		5
	Monitoring of changes in artefacts		✓	✓	✓	✓	✓	✓		✓	✓	✓		✓	✓	✓	✓	13
S2	Manual indication of impacted links	✓																1
	Monitoring of impacted links		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	15
	Impact detection rules		✓		✓	✓	✓	✓	✓		✓		✓	✓	✓	✓	✓	12
	Data Model element type		✓		✓	✓	✓				✓		✓		✓	✓		8
	Data Model element hierarchy (parent/child)		✓													✓		2
	Data Term pairs from linked artefacts							✓										1
	Data Source code structure								✓					✓				2
	Data Interaction type & sequences										✓							1
	Data 2 artefact versions													✓				1
	Data Software artefact type													✓			✓	2
O	Change type (for affected links, flag/score)	✓	✓			✓	✓	✓	✓		✓		✓	✓	✓	✓	✓	12
S3	Tool based execution of (change type specific) rules		✓			✓		✓	✓		✓			✓	✓	✓	✓	9
	Data Model element type					✓					✓				✓	✓		4
	Data Model element hierarchy (parent/child)		✓													✓		2
	Data Linked artefacts score threshold							✓	✓									2
	Data Software artefact type													✓			✓	2
	Manual use of output	✓		✓	✓		✓			✓		✓	✓					7
S4	Manual change of links	✓	✓	✓	✓	✓	✓			✓	✓	✓	✓		✓		✓	12
	Tool based link change		✓			✓		✓	✓		✓			✓	✓	✓	✓	9

* *S1: Detection of change; S2: Detection of impacted links; O: Output of detection (as input for execution); S3: Determination of necessary link change; S4: Execution of change*

* *S2 Data: for impact detection rules other than linked artefacts; S3 Data: other than linked artefacts, impact detection data & change type*

In the following, each of the 16 *TM* approaches is summarized on the basis of the characteristics shown in Tables 4.4 (linked artefacts) and 4.5 (*TM* process).[4]

In P01, Armbrust et al. [2009] describe a manual trace link maintenance approach. In the approach, authors who perform changes in sections of specification documents additionally have to assign link flags in all linked document sections, indicating the type of change in the original sections. Link flags in a section can then be processed by another author (i.e. an expert to decide about the required link change).

[4] A more detailed description of the *TM* approaches which was the source for the approach descriptions provided in this chapter can be found in the Section A.2.1 of the appendix.

In P02, Blouin et al. [2017] describe an approach in which the affected link impact detection for linked requirements and architectural models is automated. Impact detection rules are based on the linked model elements' hierarchy and the model element types to detect architectural refinements. Impact detection rules are triggered by monitoring changes on the linked artefacts. In the execution of link changes, architectural refinement-specific rules are used to maintain links automatically.

In P03, Cleland-Huang et al. [2003] describe an approach in which links between requirements and source code are maintained manually. The only automated part is in the impact detection part of the approach: by monitoring whenever a linked artefact is changed, a notification is generated and sent to the original author of the artefact. Based on this notification, the original author then has to maintain the affected links manually.

In P04, the same authors extend the approach P03. The only difference is that the notification in this approach contains hints about how to perform the necessary link change (change type).

In P05, Drivalos-Matragkas et al. [2010] describe an approach in which links between different kinds of model elements are maintained automatically using model element type-specific impact detection rules. Impact detection rules triggered by monitoring changes on linked artefacts can detect certain link change types. For a set of defined change types, there exist further rules which are then used to execute the link change automatically. For other change types, a notification is generated and the change has to be performed manually.

In P06, Fockel et al. [2012] describe an approach in which links between different kinds of model elements and textual requirements are maintained semi-automatically. To detect the impacted links, the approach uses rules which utilize the model element type and outputs a change type which is passed to the link change, which is in turn manually performed. The constraint validation rules are triggered by monitoring changes on the linked artefacts.

In P07, Gervasi and Zowghi [2014] describe an approach in which links between textual requirements are maintained automatically. For impact link detection, the approach uses rules that utilize term pairs in linked artefacts. This approach is triggered by monitoring changes on linked artefacts. Link impact detection outputs a score for artefact pairs that is based on the linked term pairs. For link change execution, rules are used along with the score to remove and add links automatically.

In P08, Ghabi and Egyed [2012] describe an approach in which links between source code and requirements are maintained automatically. The approach is triggered manually. For impact detection the approach uses a set of rules based on source code structure and existing links to calculate a numerical value which indicates a change type. For link change execution, a threshold and the previously calculated numerical value are used to remove and add links automatically.

In P09, Kleffmann et al. [2013] describe an approach in which links between sketches (of software systems) on interactive displays are maintained manually. This approach supports link maintenance in the impact detection part: whenever a user changes a linked sketch, all the affected links and linked other sketches are highlighted. Potential change to links is then performed manually by the user.

In P10, Mäder and Gotel [2012a] describe an approach in which links between UML model elements and textual requirements are maintained semi-automatically. Rules based on interaction sequences are used to detect certain change types. The change type detection rules are triggered by monitoring changes on the linked artefacts. Change type-specific rules are used to perform link changes automatically. If there are no link change rules for a detected change type, a user notification is generated and the link change is performed manually.

In P11, Maro and Steghöfer [2016] describe an approach in which links between different model elements defined in an extendable meta-model are maintained manually. Similarly to P03, the only automated part is the initial impact detection. Whenever a linked artefact is changed, a notification is generated. The execution of the link change then has to be performed manually by a user.

In P12, Paige et al. [2011] describe an approach in which links between model elements defined in different meta-models are maintained semi-automatically. Similarly to P05, model element type-specific impacted detection rules are used. Impact detection rules triggered by monitoring then changes on the linked artefacts. The output of the link impact detection is a model element-specific change type. This change type is used to manually change the links of the respective model element.

In P13, Rahimi et al. [2016] describe an approach in which links between source code and requirements are maintained automatically. This approach uses rules, two versions of the source code or requirements, a source code structure, and the artefact type to detect refactorings. Refactoring-specific rules are used to change links automatically. The rules for refactoring detection can be triggered manually or by monitoring changes on the linked artefacts.

In P14, Schwarz et al. [2010] describe an approach in which links between different model elements are maintained semi-automatically. For impacted link detection, model element type-specific rules are used. The rules for impact detection are triggered by monitoring changes in target models, followed by subsequent changes in the source models on the same model element. Link changes originating in the source models are performed automatically by rules, whereas link changes originating in target models have to be performed manually.

In P15, Seibel et al. [2012] describe an approach in which links between different model elements are maintained automatically. Impacted link detection rules use hierarchy relations between different kinds of model elements in order to determine the affected linked model elements. For each linked model element, a change type is generated. The change type is used together with model element type-specific rules

and (again) hierarchy information between model elements in order to maintain links for each model element automatically. The rules for impacted link detection can be triggered manually or by monitoring changes on the linked artefacts.

In P16, Ying et al. [2009] describe an approach in which links between source code and other software artefacts which are defined in an extendable ontology are maintained semi-automatically. Rules using artefact type data gained from an ontology are used to create change types for existing links by monitoring changes on linked artefacts. A further set of artefact type-specific rules is used to perform the link change automatically. The approach includes a set of predefined artefact type-specific link change rules. If no link change rules are defined for a certain artefact type, a user notification is created and the link change has to be performed manually.

RQ1.2: Linked Artefacts

Table 4.4 shows the linked artefacts of the *TM* approaches. These artefacts can be software artefacts in general, development standards, source code, requirements specifications in a textual format, (*UML*) model elements, and/or sketches which are hand-drawn graphically, as well as textual descriptions of software system parts.

The most frequently used artefacts in the approaches are model elements (9) and requirements (8). Almost all approaches link at least two different artefacts. The combination of model element and requirement (4) and requirement and source code (3) are the two most common combinations.

RQ1.3 Other Data Sources Used, RQ1.4 Performance of TM and RQ1.5 Approach Automation

Table 4.5 shows the characteristics of the *TM* process for the approaches, as introduced in Section 4.3.1. It distinguishes the performed actions within the steps along with the used data. The numbers of actions or data shown in the right *sum* column do not always add up to 16. The reason for this is that some approaches support multiple actions in the steps, e.g. in approach P13, step *S1* can be triggered manually and by monitoring changes in artefacts as well. For the data used in step *S2* and *S3*, only some of the approaches use additional data to the linked artefacts.

In the following, the characteristics of the 16 *TM* approaches are summarized within the four generic *TM* process steps. This comprises the answer to research questions *RQ1.3* on the artefacts used, *RQ1.4* on how trace link maintenance is performed, and *RQ1.5* on the automation degree of the *TM* approaches.

Step 1: Detection of change

13 of the 16 approaches monitor the changes on the linked artefacts. Two of these 13 approaches can be triggered manually as well. For the other three approaches, only a manual indication of changes on linked artefacts is performed.

Step 2: Detection of impacted links

15 approaches automatically detect the links involved. In one approach this detection is performed manually. 12 of these 15 approaches use rules for the impact detection. Regarding the data sources, most (8) of the impact detection rules are based on model element type-specific linkage, whereas two approaches additionally use artefact hierarchy and one of these two further uses the interaction type. Other approaches use term pairs from already linked artefacts, two versions of artefacts, and the software artefact type. For the other 3 approaches without impact detection rules, a notification with the impacted artefacts and links is generated and sent to the user to trigger the link change determination.

Data: Output of detection (as input for execution)

All approaches output the affected artefacts and links. In addition, 12 of the 16 approaches assign a change type to the affected links. This change type can indicate how the link should be maintained, e.g. with a change type such as *delete link*, or it can indicate what kind of check is to be performed on a linked artefact, e.g. *check for changes on all description texts of linked artefacts*, or it can be a numerical value which can be used later to perform the link maintenance. The four approaches without a change type generate a user notification with the impacted artefacts and links, and link maintenance is performed manually by a user. Three of these four approaches are without impact detection rules, while one has impact detection rules and generates additional instructions on how a user should maintain the links manually in its notification.

Step: 3: Determination of necessary link change

Nine of the 16 approaches determine the necessary link change automatically by a tool-based rule execution. In seven approaches the detection output is used manually.

Out of the nine approaches with rules, most (4) use model element type-specific linkage for link change determination, whereas one of them also uses hierarchy information of different artefacts. Other approaches use the software artefact type (2) or a threshold to assess an affinity score for the links calculated in the impact detection step.

Step 4: Execution of change

Four of the 16 approaches perform the link change in a completely automated way, without any required user interaction, by using detected change types for links. Five of the 16 approaches perform the link change only in cases in which a known change type has been detected. They inform the user about cases which cannot be

handled automatically by showing affected artefacts and links. In the other seven approaches, the execution of link change is performed manually by the user. In two of these seven approaches, a change type is included in the user notification, but has to be processed manually.

Regarding the techniques for impact detection and change execution rules, the approach descriptions in the publications are often presented on a conceptual level. The mentioned techniques are summarized in the following paragraph. For impact detection based on model element type and rules, constraint checks are used. Pattern recognition is used with *SCS*. Approaches which are based on the textual contents of artefacts use *IR* and *NLP* to determine textual similarity. Some of the model-based approaches store the impact detection rules in specific meta-model element attributes. Other model-based approaches transfer the model elements to a graph representation and then use graph merging and other graph-based techniques in their rules. One approach uses an ontology and reasoning to manage different software artefact types and for the purposes of implementing the impact detection rules.

RQ1.6 Performed Evaluations

Table 4.6. Performed Evaluations for Trace Link Maintenance Approaches

Evaluation Properties		P01	P02	P03	P04	P05	P06	P07	P08	P09	P10	P11	P12	P13	P14	P15	P16	∑
Research Process	Qualitative	✓	✓				✓				✓					✓		5
	– Feasibility	✓	✓				✓				✓					✓		2
	Quantitative[1]				✓			✓	✓		✓			✓		✓		6
	– Scalability/ Performance															✓		1
	– Precision				72			85	95		96			86				5
	– Recall				22			64	96		79			88				5
Data Collection	Experiment with Developers	✓									✓					✓		3
	Application with Developers										✓							1
	Questionare/ Interview	✓					✓											2
	Simulation of the approach				✓			✓	✓					✓				4

[1] *For precision and recall, the best achieved values are shown; these values could be achieved in different runs with different settings*

Table 4.6 shows whether an evaluation has been performed for an approach and the properties of the evaluation. Basic properties are the research process (qualitative/quantitative) and the data collection method used. For all approaches which were not evaluated except P16, the authors provided an running example.

For nine of the 16 approaches an evaluation was performed. The evaluations were either purely quantitative (4), purely qualitative (3) or mixed (2). The qualitative evaluations always consider the feasibility of each approach. The quantitative evaluations mostly concerned precision and recall and once only scalability and performance. Data was often collected in an experiment with developers (3) or within a simulation of the approach (4). On two occasions an interview was used and only once was the approach used in real world application with developers. For simulation and real world application, both precision and recall have been determined.

4.3.2 Summary and Discussion

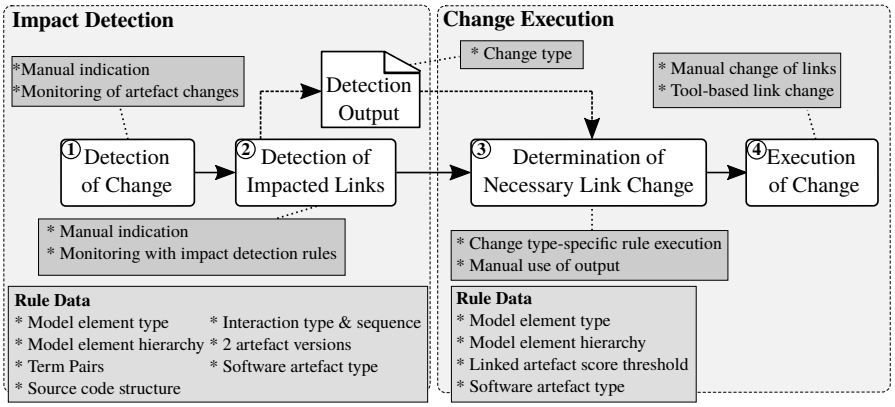

FIGURE 4.4. TRACE LINK MAINTENANCE PROCESS WITH TM APPROACH DATA

Figure 4.4 summarizes the key facts of all 16 evaluated *TM* approaches within the generic *TM* process. Monitoring of changes and monitoring of impacted links is quite prevalent. However, the data used and the techniques differ quite considerably. Also, the output of a change type is very common. Tool-based change determination is still prevalent, but it is less common than monitoring. Again different data is being used. Interestingly, only a few approaches perform changes which are purely automated. In most approaches manual change is necessary, at least for some cases. This might also be the reason why only one approach was applied in industry. The approaches are quite different. The generic *TM* process presented is the first to provide a unifying view. This can be used to integrate the ideas of different approaches. In Chapter 6, such an integration for the *ILog* approach developed in this thesis will be discussed in detail.

Part III

Treatment Design

Chapter 5

Interaction Log Recording-based Trace Link Creation

In this chapter the details of the *Interaction Log Recording-based Trace Link Creation* (*ILog*) approach are introduced. In the design cycle of this thesis, the *ILog* approach is the treatment design artefact used to solve the previously investigated problem of trace link quality for continuous trace link creation and direct usage. With this in mind the *ILog* approach is designed to primarily address the research goal **G1** of this thesis, namely to continuously create trace links with a good quality so that the created links can be directly used by developers. This also includes the goal that developers should not be interrupted during their work when *ILog* is applied and no additional efforts are necessary.

Good quality refers to the precision and recall of the links created by *ILog*. The goal of the resulting link quality of the *ILog* approach is to have perfect precision, i.e. that *ILog* does not create any wrong links, while still keeping up good recall. The core idea of *ILog* is to countervail the bad precision of existing trace link creation approaches by not directly using the contents of linked artefacts. Instead, *ILog* uses interactions of developers with source code files while they work on a requirement.

In the following, Section 5.1 gives an overview of the three steps which altogether comprise the *ILog* approach. Section 5.2 then introduces the detailed design for each step, including the technologies used and the respective rationale for them.

5.1 Overview

Figure 5.1 shows the overview of the three steps of the *ILog* approach, including the data used in the steps and the data output created by the steps. In the following these three steps are outlined:

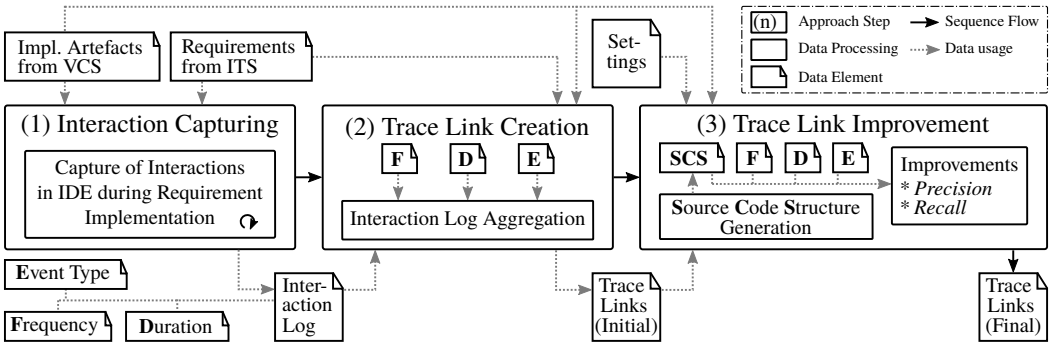

FIGURE 5.1. OVERVIEW OF THE THREE *ILog* APPROACH STEPS

(1) Interaction Capturing. In this step interactions on source code files are captured, within an *IDE* while a developer is working on a requirement, as interaction log. The recorded interaction logs are then assigned to requirements. Section 5.2.1 describes this step in detail.

(2) Trace Link Creation. In this step the recorded interaction logs are aggregated to initially created trace links, from requirements to the source code files which have been touched by interactions. The aggregated interaction log data (frequency, duration, event type, etc.) for each trace link is attached as attributes to the trace links. Section 5.2.2 describes this step in detail.

(3) Trace link Improvement. In this step the precision of trace links is improved using the interaction log data from the links and *Source Code Structure* (*SCS*) along with existing links. Recall is improved using existing links and *SCS*. All improvements can be configured. Section 5.2.3 describes this step in detail.

5.2 Details

This section introduces the details for the three steps of the *ILog* approach. In each step data is process and passed over to the next step. In the first step, interactions of developers are captures while they work in an *IDE* and they are assigned to a requirement as interaction log. In the second step, the requirement specific interaction log is aggregated to an initial set of trace links (link candidates). In the third step, the quality of the initial set of trace links is improved using the aggregated interaction data, existing links and the *SCS*.

5.2.1 Interaction Event Capturing

Figure 5.2 shows the operation principle of *ILog's* interaction capturing. A developer interacts with source code files while working in the *IDE*. The interaction events which are created in this way are recorded in an interaction log and finally assigned to a requirement. The most essential part of the interaction capturing step is the matter of how the interaction log assignment to a requirement is actually performed.

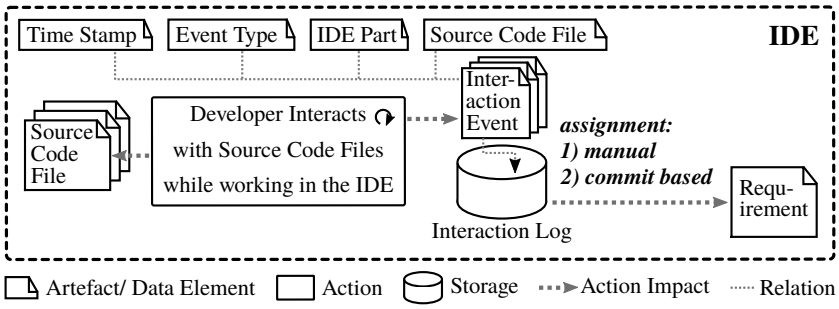

FIGURE 5.2. ILOG APPROACH STEP 1: INTERACTION CAPTURING

The interaction log assignment is crucial since in the following trace link the creation between requirements and source code is only possible if there is a connection between the recorded interaction logs and a requirement.

The *ILog* approach implements two options with which to assign interaction logs to requirements. The first option is the manual assignment by a developer. The second option is a commit-based assignment. Both of these interaction capturing options have been implemented.

5.2.1.1 Manual Assignment

The manual assignment option is inspired by the Mylyn Eclipse Plugin [Kersten and Murphy, 2006] (cf. Section 7.1.1). A developer manually indicates when starting to work on a issue and also when stopping the work. Stop indication can be performed by explicitly stopping the issue or by staring to work on another issue. Every time a stop indication is detected the interaction events recorded since the last stop indication are assigned to the indicated requirement. This simple interaction assignment principle has achieved good results regarding the interaction quality in the Mylyn Project [Soh et al., 2018]. The manual indication mode has been implemented in the *IntelliJ IDE*[1] with two IntellJ Plug-ins:

1. To log interactions the *IntelliJ Activity Tracker*[2] plug-in is used in a modified version. For *ILog* the plug-in has been extended with the ability to track the interactions with requirements. To actually use the plug-in and record interactions with it, it has to be activated once. After this, all interactions within the *IDE* of the developer are recorded, comprising a time stamp, the part of the *IDE*, and the type of interactions performed (cf. Section 2.1.4.1). The most important parts of an *IDE* regarding trace link creation from requirements to source code files are parts in which it is possible to interaction with files and parts to detect certain high-level actions of a developer. These *IDE* parts are the editor for the source code, the navigator which displays a structural tree

[1] `https://www.jetbrains.com/idea/`, IntellJ *IDE* website

[2] `https://plugins.jetbrains.com/plugin/8126-activity-tracker`, project documentation of the original Activity Tracker plug-in

of all resources managed by the *IDE*, and dialogues which are often involved in high level actions, such as committing to the Git *VCS* and performing issue-related actions. The interaction types can also be low-level interactions, such as editor keystrokes, as well as high-level interactions (selected from the context menu), such as performing a refactoring, or committing changes to a *VCS*, such as Git.

2. To associate interactions with requirements the *Task & Context* IntellJ functionality is used. In the optional initial setup of the plug-in developers can connect the plug-in to an *ITS*, such as Jira[3] and then select a project from the connected *ITS*. If a connection to an *ITS* project has already been configured, a developer can selected the specific issue with the *Task & Context* functionality when working on a requirement. When committing code changes to a *VCS* repository such as Git, the *Task & Context* plug-in supports the finishing of the respective issue. That is, after the commit has been performed, a notification message to switch to another issue if applicable is shown in the *IDE's* notification bar. As a result, the interaction log contains activation and deactivation events for requirement issues. These activation and deactivation events are used to allocate all interactions between the activation and deactivation events for a specific requirement to this specific requirement.

The following listing 5.1 shows two abridged log entries which were both created by the modified version of the Activity Tracker Plug-in.

Listing 5.1. Activity Tracker Log Entries

```
1 2016-10-04T10:14:50.910;dev2;Action;EditorSplitLine;ise;Editor;
      /git/Controller.js;
2 2016-10-13T13:28:26.414;dev2;Task Activation;ISE2016-46:Enter Arrays;
      ise;
```

The first log entry is a typical edit interaction starting with a time stamp (*2016-10-04T10:14:50.910*), the developer's user name (*dev2*), the kind of performed action (*Action*), the performed activity (*EditorSplitLine*, which is entering a new line), the name of the used Git *VCS* project (*ise*), the *IDE* part (*editor*) involved in which the interaction occurred, and the source code file (*/git/Controller.js*). The second log entry shows an interaction with a story issue from the Jira *ITS* including its issue ID and name (*ISE2016-46:Enter Arrays*). The interaction type *Task Activation* indicates that the developer *dev2* indicated to start working on the issue *ISE2016-46*.

5.2.1.2 Commit Based Assignment

Since the previously described manual assignment option is inapplicable with the principle design goal of *ILog* to create no additional effort for developers, there is a

[3]Jira is a popular *ITS* used in many open source and commercial projects. More information can be found at the Jira overview website: `https://www.atlassian.com/software/jira`

second commit-based assignment option for interaction logs. This option builds on the common software development convention to use a *VCS* and to use an issue ID in commit messages when committing changes to the *VCS* [Rath et al., 2017] (cf. Section 2.1.2). Similarly to the manual assignment, every time that a commit with an issue ID is detected, *ILog* uses the issue ID from the commit message which is provided by the developer to assign the interaction events which have been recorded since the last commit with an issue ID.

If multiple issue IDs are contained in the commit message, the recorded interactions are assigned to all issue IDs. If no issue ID is contained in the commit message, interaction recording continues until there is a commit with a commit message containing an issue ID. Clearly, this can impact precision and recall, as the commits without ID might be associated with another issue [Herzig and Zeller, 2013, Kirinuki et al., 2014].

ILog's commit based interaction capturing was implemented as a plug-in for the Eclipse *IDE*[4] and is capable of uploading recorded interactions to assigned Jira issues. Whenever a commit with an issue ID is performed, the interaction capturing tool bundles all the recorded interactions together and uploads them to the Jira issue which was specified by the Jira issue ID in the commit message. Since the tool builds on an existing *Application Programming Interface* (*API*) other *ITS* are also directly supported. The interaction events recorded by the tool comprise a time stamp, the type of interaction (select or edit), the part of the *IDE* in which the interaction occurred (e.g. editor, navigator, etc.), the file involved in the interaction, and a degree of interest (DOI) metric for the file. The DOI is a numerical value calculated for a file on the basis of the number of interactions (frequency) and the type of interactions with the file. That is, edit interactions are rated more highly than select interactions [Kersten and Murphy, 2006]. Expect the more coarse interaction type and the DOI value, the interaction attributes are identical in the commit based and in the manual recording tools. The interaction logs assigned to issues (requirements) are the data output created by the first *ILog* step. This data is passed and then further processed by the second *ILog* step which is described in the following.

5.2.2 Trace Link Creation

In the second *ILog* step *Trace Link Creation*, all interaction events captured for a requirement by the previous step are used to generate trace links between the requirement and the source code files which are touched by the interactions.

Initially a check is performed to examine whether the issue ID used in the interaction log actually refers to a requirement. If this is not the case and the issue ID refers to another issue type such as a work item, the linkage of the work item issue in the *ITS* is used to identify the requirement which the work item and thus the

[4] `https://www.eclipse.org`, Elipse *IDE* website

recorded interaction log belongs to. After this initial check all recorded interactions are assigned to real requirements.

To avoid the involvement of interactions, which are probably not relevant for trace link creation, three different filter options can be applied before the trace links are actually created. Firstly, it is possible to exclude interactions with files of certain types. Secondly, the files from the interactions can be aligned with files in a *VCS*. And thirdly, interaction with certain interaction types can be removed.

By excluding interactions with files of certain types such as build configurations, project descriptions, readme files, meta-data descriptions, binaries etc. as well as files from third parties such as libraries, it is possible to focus the resulting links on the code created by the developers. Aligning interaction with files in the *VCS* results in removing links to temporary files and files which are only kept locally by a developer. Naturally such local and temporary files would not be accessible by other developers when following a trace link to such a file. Furthermore, for such files it is unlikely that they contribute to a requirement, which is to say that a link to such a file has a high probability of being wrong. Removing interactions of certain types is necessary since there are interactions which do not contain a source code file: e.g. if a developer performs a change in the *IDE* configuration, which took place in a part of the *IDE* that was not relevant for trace link creation; or involving the console of the *IDE*, or generated interactions which are not directly triggered by a developer; or if a build file of a project is changed, then this can cause the *IDE* to rebuild the project. In this case the recorded rebuild interaction is not relevant for trace link creation.

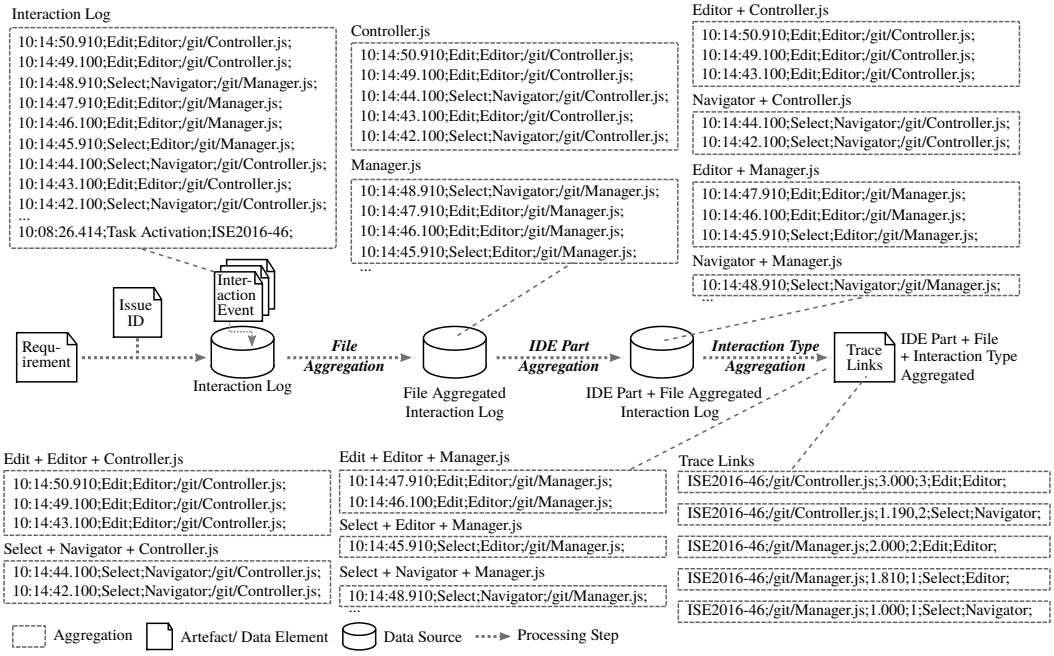

Figure 5.3. ILog Approach Step 2: Trace Link Creation by Interaction Aggregation

After this step, the initial filtering of interactions trace links are generated by aggregating the interaction events. Figure 5.3 shows the process of *ILog's* interaction aggregation along with exemplary data which is used for the following explanation. Basically it is also possible to configure the interaction aggregating. As shown in the sample interaction log excerpt on the left side of Figure 5.3, interaction events in the interaction log are present in descending chronological order (latest on top).

Firstly, all interactions to the same file are aggregated. Secondly, the interactions for the same file are aggregated by the *IDE* part, and thirdly by the interaction type. Along with the three aggregations the frequency and duration values are calculated. The frequency is calculated by counting the number of interactions involved after the third aggregation step. The duration is calculated as the sum of all differences from the interactions involved after the third aggregation step to the previous interaction from the original interaction log. The final result of this second step is a list of trace links including the data aggregated from the interactions for each link, which is used as an input for the third *ILog* step *Trace Link Improvement*.

The second *ILog* step has been implemented in two ways. First it was implemented in a Python *NLTK*[5] (Natural Language Toolkit)-based tool, which was used in the performed evaluation studies. Second it was implemented as a plug-in for the Jira *ITS*, which directly enables one to provide *ILog* generated links to developers. In addition to the described aggregation process, its configuration options and the link creation, the implementations are also capable of reading the interaction data from both interaction capturing tools.

5.2.3 Trace Link Improvement

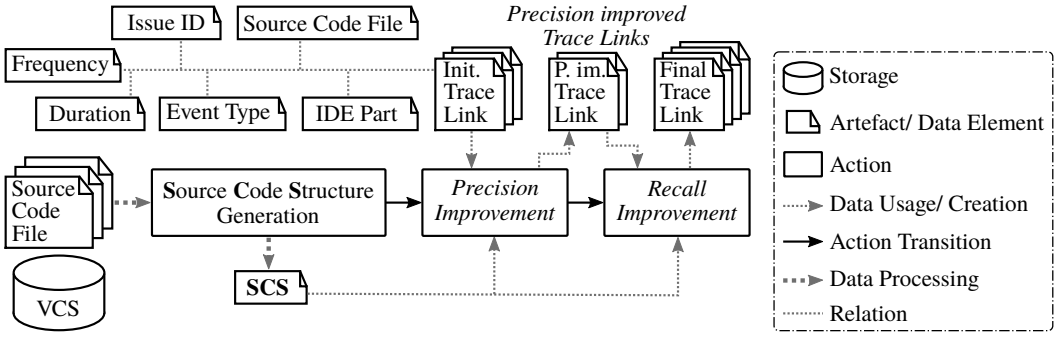

Figure 5.4. ILog Approach Step 3: Trace Link Improvement

Figure 5.4 shows the overview of *ILog's trace link improvement* step and the data sources used. Precision is improved by removing potential wrong links using the interaction-specific data attributes *frequency, duration, event type* and *IDE part* from the previous link creation step. For all attributes different settings are possible. Precision is also improved by using the previously generated *SCS*, i.e. the references

[5] `https://www.nltk.org/`, *NLTK* documentation website

from one source code file to other source code files. *SCS* is used to remove links from requirements to source code files which are not connected to the other source code files that are linked to the same requirement.

Finally *SCS* is also used to improve the recall of *ILog*. In this case, the *SCS* of source code files which are already linked to a requirement is utilized. New trace links are added by following the relations of the *SCS* to other source code files up to a certain level. In the following the single improvement techniques are described in detail along with an example. After this the combination of the improvement techniques is described.

5.2.3.1 Precision

In order to improve the trace links created in the previous trace link creation step, it is necessary to first apply precision improvements to remove potentially wrong links. To avoid negative impacts on the precision, recall improvement is only reasonable after precision improvements are finished. The reason for this is that recall improvements use existing links as input and if these existing links are incorrect, the precision would be impaired.

Interaction Log

Listing 5.2. Initial Trace links Created in ILog's Trace Link Creation Step

```
1 ISE2016-46;/git/Controller.js;Select;23.710;12;Editor;
2 ISE2016-46;/git/Controller.js;Edit;421.590;31;Editor;
3 ISE2016-46;/git/Manager.js;Select;9.370;27;Navigator;
```

The Listing 5.2 shows an example of three trace links as created by *ILog's* previous trace link creation step. The format of the link's attributes is:

<Requirement ID; Soruce Code File; Interaction Type; Duration; IDE Part;>

Requirement ID and *Soruce Code File* denote the source and target of the trace links. The other attributes are the aggregated interaction specific data. Thus the link in line 1 is a link from the requirement with issue ID *ISE2016-46* to the source code file */git/Controller.js*. The link was created based on an interaction of the type *Select* with an duration of *23.710* seconds and the *Editor* part of the *IDE*.

Listing 5.3. Trace Links With Applied Duration Precision Improvement

```
1 ISE2016-46;/git/Controller.js;Select;23.710;12;Editor;
2 ISE2016-46;/git/Controller.js;Edit;421.590;31;Editor;
3 ISE2016-46;/git/Manager.js;Select;9.370;27;Navigator;
```

For *ILog's* duration-based precision improvement, links with a duration below a certain threshold are removed. The idea behind this is that links resulting from

short durations are more likely to be wrong, e.g. if a developer started to implement something in the wrong source code file but changed to the correct file after a short time period. In the example shown in Listing 5.3 the duration threshold is set to *>=10.000 sec.* and thus the link in line 3 is removed, since its duration is only *9.370* sec.

Listing 5.4. Trace Links With Applied Frequency Precision Improvement

```
1 ISE2016-46;/git/Controller.js;Select;23.710;12;Editor;
2 ISE2016-46;/git/Controller.js;Edit;421.590;31;Editor;
3 ISE2016-46;/git/Manager.js;Select;9.370;27;Navigator;
```

Also for *ILog's* frequency-based precision improvement, links with a frequency below a certain threshold are removed. The idea behind this is that links resulting from a low frequency are more likely to be wrong, e.g. if a developer selects a file by accident. In the example shown in Listing 5.4 the frequency threshold is set to *>=15* and thus the link in line 1 is removed, since its frequency is only *12*.

Listing 5.5. Trace Links With Applied Event Type Precision Improvement

```
1 ISE2016-46;/git/Controller.js;Select;23.710;27;Editor;
2 ISE2016-46;/git/Controller.js;Edit;421.590;31;Editor;
3 ISE2016-46;/git/Manager.js;Select;9.370;27;Navigator;
```

For *ILog's* event type-based precision improvement, links with certain event types are removed. The idea behind this is that links resulting from certain event types are not relevant; for example, if a developer browses code and selects multiple source code files in doing so, it is unlikely that all browsed files should be linked with the actual implemented requirement. Excluding links of the event type *Select* would prevent the link creation to all files browsed by the developer. In the example shown in Listing 5.5 links of the event type *Select* are excluded and thus the links in line 1 and 3 are removed.

Listing 5.6. Trace Links With Applied *IDE Part* Precision Improvement

```
1 ISE2016-46;/git/Controller.js;Select;23.710;27;Editor;
2 ISE2016-46;/git/Controller.js;Edit;421.590;31;Editor;
3 ISE2016-46;/git/Manager.js;Select;9.370;27;Navigator;
```

ILogs's IDE part-based precision improvement is similar to the event type-based improvement, links with a certain *IDE* part are removed. The idea behind this is that links resulting from certain *IDE* parts are not relevant, e.g. if a developer browses code by selecting multiple files in a row in the navigator part of the *IDE*, it is unlikely that all files browsed through the navigator should be linked with the actual implemented requirement. Excluding links with *IDE* part *Navigator* would prevent the link creation to all files browsed by the developer through the navigator. In the example shown in Listing 5.6 links with the *IDE* part *Navigator* are excluded and thus the link in line 3 is removed.

Source Code Structure

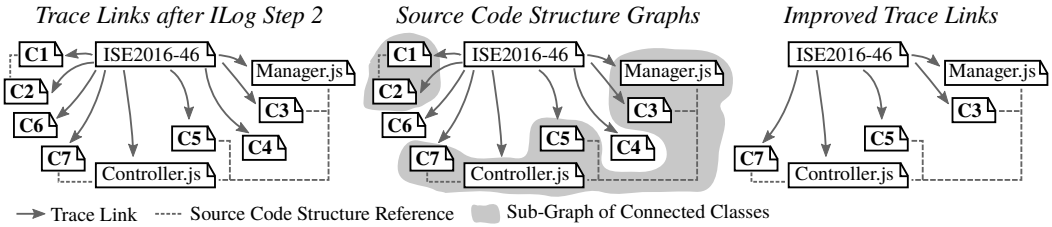

Figure 5.5. ILog Approach Step 3: Source Code Structure Precision Improvement

ILog's Source Code Structure (*SCS*)-based precision improvement utilizes the references between source code files involved in the trace links of one requirement. Details of *SCS* and usage of *SCS* in traceability were introduced in Section 2.1.3. The right side of Figure 5.5 shows the trace links between the requirement *ISE2016-46* and the classes *Controller.js, Manager.js* and *Cn* and the *SCS* (i.e. the references) between the classes.

Listing 5.7. Trace Links Before Source Code Structure Based Precision Improvement

```
1  ISE2016-46;/git/Controller.js;Edit;421.590;31;Editor;
2  ISE2016-46;/git/Manager.js;Select;9.370;27;Navigator;
3  ISE2016-46;/git/C1.js;Edit;421.590;31;Editor;
4  ISE2016-46;/git/C2.js;Select;33.940;3;Editor;
5  ISE2016-46;/git/C3.js;Edit;1.500;17;Editor;
6  ISE2016-46;/git/C4.js;Edit;91.300;231;Navigator;
7  ISE2016-46;/git/C5.js;Edit;17.340;3;Editor;
8  ISE2016-46;/git/C6.js;Select;21.120;91;Console;
9  ISE2016-46;/git/C7.js;Edit;42.230;1;Editor;
```

Listing 5.7 shows the trace links for the situation as shown on the right side of Figure 5.5 after their initial creation in *ILog's* second step, before the application of the *SCS*-based precision improvement. As shown in Figure 5.5, the classes *Controller.js, Manager.js, C3, C5* and *C7* and the classes *C1* and *C2* are connected by *SCS*. In the middle of Figure 5.5 the *SCS* sub graphs created by both groups of classes are highlighted. As shown on the right side of Figure 5.5 the *SCS* precision improvement then removes all links to classes which are not part of the largest *SCS* sub graph.

Algorithm 1 shows the *SCS* precision improvement algorithm of *ILog*, which performs the sub graph creation and the removal of all links to classes that are not part of the largest sub graph. It uses a requirement with its trace links and the *SCS* graph of all source code files (classes) as input. The algorithm consists of two parts.

First (cf. line 2 to 16 of Algorithm 1), the largest requirement specific *SCS* sub graph is created by iterating through all classes linked by trace links to the requirement. For each trace link of the requirement the trace link specific *SCS* graph reachable for each linked class is generated. The trace link specific *SCS* graphs are compared so that only the largest (i.e. the one which contains the most classes) of the graphs is kept.

Algorithm 1: ILogs's Source Code Structure based Precision Improvement

```
   input : SCS: Source code structure graph consisting of a list of classes C
           Req: Requirement with trace links Req.tls
   output: Req: Requirement with improved trace links Req.tls
 1 function SCSprecisionImprovement(Req, SCS) :
 2     // create the largest SCS sub graph lSCS for Req
 3     list lSCS
 4     foreach C of tl in Req.tls do // iterate linked classes
 5         list tSCS
 6         while hasReferences(C) do // iterate referenced classes
 7             // append referenced classes to the SCS tSCS of tl
 8             refC = nextReference(SCS,C)
 9             if refC not in tSCS then
10                 tSCS append refC
11             end
12             C = refC
13         end
14         if tSCS > lSCS then
15             lSCS = tSCS
16         end
17     end
18     // remove links tl from Req that link to classes not in lSCS
19     foreach tl in Req.tls do
20         if C of tl not in lSCS then
21             Req.tls remove tl
22         end
23     end
24     return Req
25 end
```

Second (cf. line 18 to 22 of Algorithm 1), all trace links of the requirement that link to classes which are not contained in the largest requirement specific *SCS* sub graph are removed and the updated requirement is returned.

For the example shown in Figure 5.5 and in Listing 5.7 the *SCS* based precision improvement removes the links to the classes *C1, C2, C4* and *C6* as shown in Listing 5.8 (removed links are highlighted) and on the right side of Figure 5.5.

Listing 5.8. Trace Links After Source Code Structure Based Precision Improvement

```
1  ISE2016-46;/git/Controller.js;Edit;421.590;31;Editor;
2  ISE2016-46;/git/Manager.js;Select;9.370;27;Navigator;
3  ISE2016-46;/git/C1.js;Edit;421.590;31;Editor;
4  ISE2016-46;/git/C2.js;Select;33.940;3;Editor;
5  ISE2016-46;/git/C3.js;Edit;1.500;17;Editor;
6  ISE2016-46;/git/C4.js;Edit;91.300;231;Navigator;
7  ISE2016-46;/git/C5.js;Edit;17.340;3;Editor;
8  ISE2016-46;/git/C6.js;Select;21.120;91;Console;
9  ISE2016-46;/git/C7.js;Edit;42.230;1;Editor;
```

Further source code specific precision improvements are possible with *ILog*. The *prominent code files in requirement* improvement only uses files which are used in

multiple requirements. The *source code type* improvement limits the used source code files to a list of specific source code file types such as Java, JavaScript, XML etc. However these further improvements did not have a positive effect on the precision of *ILog's* trace links in the evaluation studies (cf. Chapter 9) and thus are discussed in detail there.

The *SCS* based precision improvement has been implemented with the Esprima[6] JavaScript source code parser for JavaScript source code files and with Eclipse *JDT*[7] for Java source code files in the Python-based implementation of *ILog*.

5.2.3.2 Recall

ILog's recall improvement capabilities add further links from requirements to source code files by using existing links. To avoid a negative effect on the precision of *ILog's* links as far as possible, the recall improvement is always applied after all precision improvements.

Source Code Structure

Algorithm 2: ILogs's Source Code Structure based Recall Improvement

```
   input : SCS: Source code structure graph consisting of a list of classes C
           Reqs: Requirements list with requirement Req links Req.tls
           Limit: Limit for the SCS graph traversal level
   output: Reqs: Requirements with improved requirement Req links Req.tls
 1 function SCSrecallImprovement(Reqs, SCS, Limit) :
 2     foreach Req of Reqs do // iterate requirements
 3         foreach C of tl in Req.tls do // iterate linked classes
 4             Level = 1
 5             while hasReferences(C) do // iterate referenced classes
 6                 // create new trace link from Req of refC
 7                 refC = nextReference(SCS,C)
 8                 if [Req, refC ] not in Req.tls then
 9                     Req.tls append [Req, refC ] // add new trace link
10                 end
11                 if Level ≥ Limit then
12                     break // stop if limit is reached
13                 end
14                 C = refC
15                 increment Level
16             end
17         end
18     end
19     return Reqs
20 end
```

[6] `https://esprima.org/`, *Esprima* documentation
[7] `https://www.eclipse.org/jdt/`, Eclipse *Java development tools (JDT)* documentation

In addition to precision improvement, *SCS* is used together with existing links for recall improvement as well. Algorithm 2 shows *ILog's* recall improvement algorithm. It uses the *SCS*, the list with all requirements, and a limit for the *SCS* graph traversal level as input. The algorithm iterates through all classes (source code files) linked to requirements. For each class the *SCS* graph is traversed and the referenced classes are visited (cf. line 5 to 15 in Algorithm 2). If there are no trace links between the referenced classes and the requirement, new trace links from the requirement to the referenced classes are added until the specified traversal level limit is reached. After the processing of all links of all requirements, the algorithm returns the list with requirements with updated, i.e. newly added, links.

The implementation of the *SCS*-based recall improvement reuses the implementation of the *SCS*-based precision improvement (cf. Section 5.2.3.1) for source code file parsing and *SCS* graph generation.

5.2.3.3 Combined

When *ILog* is used for trace link creation in a project the different improvement techniques are applied in combination. The only fixed setting in *ILog's* implementation is to apply all precision improvements before improving recall (cf. Figure 5.4 for the sequence for improvement application). The order and settings for the different precision improvement techniques are freely configurable. In the *ILog* evaluation studies (cf. Chapter 9 and 10) performed, the setting combinations to achieve the overall best precision improvements have been determined empirically and are discussed in more detail there.

Chapter 6

Integration of Trace Link Maintenance

This chapter contains the second part of the treatment design task of this thesis and primarily addresses research goal **G2**. In this chapter the results of the *SLR* about *Trace Link Maintenance* (*TM*), from the previous Chapter 4, are used to discuss the integration of *TM* capabilities into the *ILog* approach (cf. Chapter 5).

In Section 6.1 the selection criteria for suitable *TM* approaches are introduced and applied to select the two *TM* approaches P08 and P13. In Section 6.2, the integration of the selected *TM* capabilities into *ILog* is discussed.

6.1 Approach Selection

Table 6.1. Selection Criteria and Trace Link Maintenance Approach Rating for Integration of Maintenance Capabilities in *ILog*

	Criteria	Description	Approach Rating		
			P03	P08	P13
C1	Linked artefatcs	Linked artefacts have to be the same as in *ILog* *(requirements, source code)*	✓	✓	✓
C2	Additionally used data sources	Additionally used data sources or techniques to maintain links have to be available in *ILog* (interaction logs, source code structure)	✓	✓	✓
C3	Automation	Impact on the overall automation of *ILog* *(Impact detection and link change should be fully automated)*		✓	✓

Table 6.1 shows the overview of the selection criteria, their description and the rating of *TM* approaches for the criteria.

The first selection criterion (C1) for the transferability of a *TM* approach to *ILog* is the match of the source and target artefacts used for links. The second selection criterion (C2) is the availability of the additional data sources and techniques. The third selection criterion (C3) is the degree of automation provided by an *TM* approach and the influence on *ILog's* automation. These criteria enable a smooth integration of *TM* capabilities into *ILog*. Only the three *TM* approaches P03, P08

and P13 shown in Table 6.1 satisfy the criteria C1 (match of linked artefacts). Thus, the right sided column of Table 6.1 only shows the rating of the selection criteria for these three *TM* approaches.

Out of these only the approaches P08 and P13 satisfy all selection criteria. In P03, the determination of impacted links and the execution of link changes are manual. Therefore, C3 is not satisfied. The change detection in P08 is manual, but the other steps are automated. As the change detection can easily be automated in the *ILog* approach, C3 can be considered satisfied. P13 is fully automated and thus satisfies C3.

The approach P08 uses *SCS* in their rules for impacted link detection and thus satisfies C2. The approach P13 requires a second version of source code files or requirements in addition to linked artefacts. Due to the usage of an *ITS* and a *VCS* along with *ILog* when using commit-based interaction assignment, a second version of source code files and requirements is directly available. Thus, the approach P13 also satisfies C2.

Therefore, in the following it is discussed how the techniques used in P08 and P13 can be integrated into *ILog*.

6.2 Integration of Trace Link Maintenance Capabilities

In this section a *TM* process, using the generic *TM* process of the *TM SLR* (cf. Section 4.3), for *ILog* when using commit-based interaction assignment is defined, based on the data and techniques used in P08 and P13.

S1: Detection of change

After each commit there is an automated check of the interactions whether an artefact was changed. A change is indicated by an edit interaction. This corresponds to the manual change detection of P08 and P13.

S2: Detection of impacted links

The detection of impacted links is performed by the rules from P08 and P13. As discussed for criterion C2, the data sources are available. In addition to using two artefact versions for the detection of refactorings, as in P13, in *ILog* it is possible to use the recorded interactions from *S1*. *IDEs* provide capabilities to perform certain refactorings, such as the refactoring *extract method to class*. Such refactorings can be directly detected in the interactions. Furthermore, one could define patterns of low level interactions which comprise the interaction sequence of a certain refactoring. These patterns could also be detected automatically in the interactions.

Data: Output of detection:

The rules used in *S2* output a change type.

S3: Determination of necessary link change and S4: Execution of change:

The determination of link changes and the execution of the changes is fully automated for P08 and P13. Therefore these steps are performed in a fully automated way.

The link changes derived from the affinity scores used in P08 are similar to the link changes in the improvement techniques of *ILog*. The *ILog SCS*-based precision improvement removes existing links to the source code which is not connected with other linked source code (cf. Section 5.2.3.1). Similarly, P08 removes links if the affinity score of a method drops below a threshold because it is not connected by *SCS* to other linked methods. The *ILog SCS*-based recall improvement adds links by following the call relation to other source code files from source code files already linked to requirements. Similarly, P08 adds links if the affinity score of a method rises above a threshold because it is connected by *SCS* to other methods which are already linked.

Altogether, *TM* can be performed in a fully automated way in *ILog*. However, neither P08 and P13 provide an implementation. Thus, considerable effort is necessary to implement the process described above. Therefore, the implementation is not part of this thesis.

Part IV

Treatment Validation

Chapter 7

Overview of Evaluation Studies

In this chapter the data acquisition for the evaluation of the *ILog* approach is described and an overview of the evaluation studies which were performed is presented. To this end Section 7.1 introduces the projects which have been used to collect the data. Section 7.2 describes how the data collected in the projects was processed in order to create the datasets used in the evaluation studies. Section 7.3 introduces the creation process and the resultant gold standards used in the evaluation studies. Section 7.4 introduces the evaluation toolset which was used to create and compare trace links with the different trace link creation techniques and settings of the studies. Section 7.5 summarises all trace link creation techniques used in the evaluation studies which were performed. Finally, Section 7.6 presents an overview of the three evaluation studies which were performed, shows how the created datasets were used within the studies, and summarizes the studies' different characteristics.

7.1 Evaluation Projects

The *ILog* approach has been evaluated with data collected from three projects, which are introduced in the following paragraphs. Section 7.1.1 introduces the Mylyn[1] open source project. Section 7.1.2 and Section 7.1.3 introduce the 2017 and 2018 student internship projects, respectively.

7.1.1 Mylyn

Mylyn is a plug-in (i.e. an extension) of the Eclipse[2] *IDE* which enables the management of development tasks directly in the *IDE*. The open source development of Mylyn began in 2005, after it was initially developed as a research prototype [Kersten and Murphy, 2006]. Mylyn records the interactions of developers with artefacts, assigns them to tasks and then is able to limit the views in the *IDE* to contain only

[1] `https://www.eclipse.org/mylyn/`, Mylyn project website
[2] `http://www.eclipse.org`, Eclipse *IDE* website

artefacts for which interactions have been recorded. The goal of Mylyn is to reduce the information overload in large development projects. The Mylyn developers call this principle a *task-focused UI*.

The tasks managed by Mylyn can be associated with issues from an *ITS* by connecting the *IDE* to an *ITS*, such as Bugzilla[3], Jira, etc. When the interaction recording of Mylyn is activated, Mylyn logs edit-, select- and other events after a developer has selected an issue from an associated *ITS*. Interactions concerning an issue are recorded until the developer finishes working on the issue, e.g. by closing the issue, by switching to another issue, or by explicitly stopping the recording of interactions.

For the development of Mylyn the developers use Mylyn together with the *ITS* Bugzilla[4], which is also used for requirements capture. The Mylyn developers are encouraged to trigger recording when they work on the implementation of a requirement. These interaction logs are accessible as attachments of the issues in the Mylyn Bugzilla *ITS*. Since Mylyn is an Eclipse plug-in, its source code is mainly written in Java. In the project, the developers use Git as *VCS* to manage the source code. The Mylyn Git repository[5], and thus all Mylyn source code files, are publically available.

The Mylyn projects provides all data which is required to evaluate the *ILog* approach, including interaction logs, requirements and source code. For the evaluation of *ILog*, two datasets using excerpts of the Mylyn project data from 2007 and 2012 were used. The creation of these two datasets, which are called M_{2007} and M_{2012} respectively, is described in Section 7.2.2.

As already introduced in Section 2.1.4.2, the interaction log data of the Mylyn project has also been used by other researchers for different research purposes, in order to judge the correctness of recorded interaction data [Soh et al., 2018], to find hidden relations between source code files [Konôpka and Navrat, 2015] and to find tasks which are similar to each other [Maalej and Ellmann, 2015].

7.1.2 Student Internship 2017 – Healthcare

In the following paragraphs, the first of the two student projects which were used for the evaluation of *ILog* is described. Due to the labour intensity of creating a trace link gold standard, student projects are often used for trace link evaluations [De Lucia et al., 2007]. The project lasted from October 2016 to March 2017 and was performed Scrum oriented [Schwaber and Beedle, 2001]. Therefore, it was separated into seven sprints with the goal of obtaining a working product increment in each sprint. Since the project finished in 2017, the dataset from this project which was

[3]Bugzilla is an *ITS* developed as open source software. More information can be found at the Bugzilla website: `https://www.bugzilla.org/`.

[4]`https://bugs.eclipse.org/bugs/describecomponents.cgi?product=Mylyn`, The Bugzilla *ITS* of the Mylyn project

[5]`https://git.eclipse.org/c/mylyn/`, Mylyn Git *VCS* repository

used for the evaluation of *ILog* is called S_{2017}. The project's aim was to develop a so-called *master patient index* for an open ID-oriented organization of healthcare patient data. A typical kind of use case for the resulting product would be to store and manage all healthcare reports for a patient in a single database. The project involved the IT department of the University Hospital as a real-world customer. Further roles which were involved were the student developers and a research group member who had the role of a product owner. Seven developers participated in the project in total. In each of the sprints one of the developers acted as a scrum master.

All requirement-related activities were documented in a Scrum project of the Jira *ITS* used. This included the specification of requirements in the form of user stories and the functional grouping of the requirements as epics. For instance, the epic *Patient Data Management* comprised user stories such as *View Patients* and *Search Patient Data*. Complex stories in turn comprised sub-tasks which documented more and often technical details and the partial work to implement a user story. For instance, the *Search Patient Data* user story comprised the sub-tasks *Provide Search Interface* and *Create Rest Endpoint*. The project started with an initial vision of the final product from the customer and was then broken down by the developers, using the scrum backlog functionality of Jira to a set of initial stories which evolved during the sprints.

For the project's implementation *JavaScript* was used, a choice which was requested by the customer. Furthermore, the MongoDB[6] NoSQL database and the React[7] UI framework were used. The developers used the Webstorm[8] version of the IntelliJ *IDE* along with Git as *VCS*. Within the Jira project and the JavaScript source code, the feature management approach of Seiler and Paech [2017] was also applied. A feature in this project corresponded to an epic. The feature management approach ensures that all artefacts are tagged with the name of the feature that they belong to. The purpose of this is that a user story is tagged with the epic it corresponds to, but also that the sub-tasks of the user stories and the code implementing the user story are tagged. The usage of this feature management approach is relevant in the context of *ILogs's* evaluation since the feature tags were used to created the developer-specific questionnaire for the gold standard creation (cf. Section 7.3).

The developers installed and configured the IntelliJ interaction-capturing tool of *ILog* (cf. Section 5.2.1.1), which was used for interaction recording and the developers were supported whenever needed. They received a short introduction about interaction recording and how requirements and source code files are associated with each other by that. The interaction-capturing tool recorded all interactions in the *IDE* in locally stored CSV and XML files. The developers were asked to send us their interaction log files by email after each sprint on a voluntary basis, so that

[6] `https://www.mongodb.com/`, MongoDB website
[7] `https://reactjs.org/`, React website
[8] `https://www.jetbrains.com/webstorm/`, Webstorm *IDE* website

it was possible to check the plausibility of the recorded interactions. In the first sprints some of the developers had problems with activating the interaction recording and using the desired IntelliJ functionality to interact with requirements. After detecting such problems, further assistance and instructions were provided to the developers and they were asked to solve these problems for the processing of the next sprint. However, some of the developers only sent their interaction logs once or twice in the final project phase. Therefore, four of the seven log files received were not usable for *ILog's* evaluation. One was almost empty due to technical problems, whereas in the other three only a very low number of requirements was logged. The corresponding developers stopped recording changes to requirements at a certain point in time and thus all the subsequent interactions were associated with the last activated requirement. Thus, only the three correctly recorded interaction logs have been used to apply and evaluate the *ILog* approach.

7.1.3 Student Internship 2018 – Indoor Navigation

In the following paragraphs the second of the two student projects used for the evaluation of *ILog* is described. The basic characteristics of the second student project are quite similar to the first one. The project took part as a student internship, was organized as Scrum-oriented working with a real-world customer in sprints, and took place between October 2017 and March 2018. Since the project was finished in 2018, the data set from this project is called S_{2018}.

The aim of the S_{2018} project was to develop an Android-based indoor navigation app for students in University buildings. Typical use cases for such an app are to navigate to the room of a certain lecture or to find any other point of interest efficiently. The customer was a mobile development company. As in the first student project, a research group member was involved as an adviser. Six students participated in the project. The project was again split into a corresponding number of sprints. In each of these sprints, one of the students acted as Scrum master and thus was responsible for all organizational concerns, such as planning the development during the sprint and communicating with the customer.

For all requirement management-related activities, a Scrum Jira Project was used. This included the specification of requirements in the form of user stories and the bundling of the user stories in epics. An example of a user story in the navigation app project is *Show point to point route*, in which case the corresponding epic of this user story is *routing*. To assign the implementation of user stories to developers, sub-task issues were used. A sub-task comprises partial work to implement a user story, e.g. *Show route info box*. For the implementation, the developers used Git as *VCS* and the Eclipse *IDE* with the Android *Software Development Kit* (*SDK*). For the recording and assignment of interactions, the students used the *ILog* commit-based assignment tool implemented as a plug-in for the Eclipse *IDE* (cf. Section

5.2.1.2). For the usage of Git, there was an explicit guideline to use a Jira Issue ID in any commit message in order to indicate the Jira Issue associated with it.

In the project the customer provided a proprietary Java *SDK* of their own for the general use case to develop Android mobile navigation apps. The developers needed two sprints to understand the complexity of the *SDK* and to set up everything in such a way that they could work efficiently on the implementation of requirements. The programming language for the logic and data management part was Java and the UI was implemented in Android's own XML-based language. As in the first student project, in the second student project the developers were also supported with the installation and initial configuration of the interaction log recording tool at the beginning of the first sprint. The student developers received a short introduction about the implemented interaction-recording mechanism and how to use the tool during the project. Since the tool also uploaded the recorded interaction logs automatically to the respective Jira Issue, contrary to the first S_{2017} student project, interaction recording from all six developers in the S_{2018} project could be used for the evaluation of *ILog*. Further regular inspections of the resultant links were performed to observe whether the students applied the approach in a disciplined way.

7.2 Data Processing

In this section the processing of data collected from the previously described projects for the purposes of performing the *ILog* approach evaluation studies is described. It consists of general data processing steps as performed for all three projects and of project-specific data processing steps.

7.2.1 General Alignment of Interactions and Source Code

All three projects used a *VCS* for source code management and an *ITS* for requirement specification, and all recorded developer's interactions while they were implementing requirements in an *IDE*.

To limit the recorded interactions and for the creation of a gold standard for created trace links, an alignment step was performed. All recorded interactions were filtered to contain only interactions with files which were also present in the project's *VCS*. This step removes interactions to files which were only present locally for a developer. The reason for this was that files which are not committed to the *VCS* are most likely not relevant for the implementation of a requirement. The recorded interactions were further filtered to refer only to files from the version of source code files created with the final commit of a project. The reason for this was that files which are not present in the final version of the implementation also do not contribute to the final version of requirements. The source code files of the final version in the *VCSs* of the projects were also filtered to contain only those files

which were directly created by the developers of the projects. This file filtering was important for the gold standard creation, inasmuch as developers can only vet links to such source code files in a reasonable way and *IR* can only create links to such textual artefacts in a reasonable way and make the comparison between *IR* and *ILog* possible.

7.2.2 Mylyn

For the creation of the two datasets M_{2007} and M_{2012}, excerpts of the Mylyn project data were used. Both datasets used for the initial evaluation of the *ILog* approach consist of data from the Bugzilla *ITS* for requirements and interaction logs and from the Git *VCS* for implementation artefacts taken from the Mylyn open source project (cf. Section 7.1.1). For trace link creation with the *ILog* approach, all three data sources (requirements, implementation artefacts, interaction logs) were used, whereas for *IR*-based trace link creation, only requirements and implementation artefacts were used. Issues in the Mylyn project have been created from early 2005, yet the open source development of Mylyn really began at the beginning of 2007 when its source code was first made publicly available. The development activity of Mylyn has decreased in the last few years but is still ongoing. One reason for this is that the major features are already implemented and development efforts are mostly concerned with bug fixing.

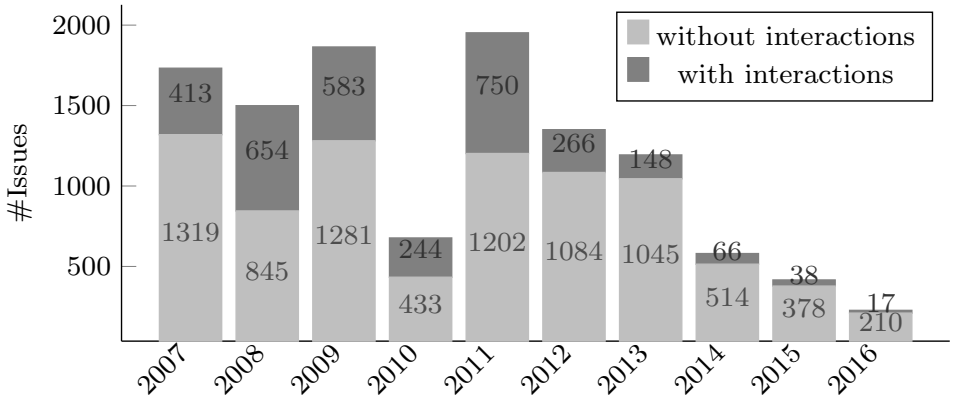

Figure 7.1. Issue in the Mylyn Bugzilla Issue Tracker System per Year

Figure 7.1 shows the number of issues with and without interactions per year until mid of June 2016 when the data was fetched. Till then there were a total of 11490 issues from which 3179 (27.7%) have interaction logs attached and therefore are suitable for the evaluation of the *ILog* approach. In total the 3179 issues have over 3 million recorded interaction events attached. Based on these general data characteristics it has been decided to evaluate only a subset of the existing interaction logs by selecting a suitable subset of requirements issues. The following criteria have been used for the requirements selection:

C1: There should be two distinct data sets from different project phases, namely from early phase and from later phase. The reason for this is to check whether *ILog* trace link creation is applicable for different project circumstances.

C2: The number of interactions in the two sets should be as similar as possible to ensure the comparability of the two datasets. Due to the data characteristics, this criterion could only be fulfilled to a certain extent, since the number of interactions by issue also decreases during the years.

These criteria resulted in the creation of two datasets. The *first dataset* M_{2007} consists of the first 50 requirements issues in 2007 (together with the corresponding interaction log and code) and the *second dataset* M_{2012} consists of the first 50 requirements issues in 2012 (together with the corresponding interaction logs and code). The first requirements of the years have been used, since the Mylyn project employs an annual release cycle with a major release every June. Therefore, new requirements are mostly created at the beginning of a year, whereas around the release date more bugs are created. Requirements are described as natural language text, using the Bugzilla issue format, in which there is a title, a description text and technical meta-data such as the affected components and the assignee (cf. Section 2.1.1). For each dataset requirement issues, including the comments and interaction logs, have been downloaded from Bugzilla using the evaluation tool (cf. Section 7.4). Comments have been included, since they often contain information relevant for the requirements, e.g. relating to changes to the functionality as it was initially stated in the description. Since there is no explicit classification of the issues either as requirement or as bug, this classification of the issues has been performed by the researchers of the study. First an overview list with all issue titles has been collected and then the classification of the issues has been performed manually by reading their title. If classification was not possible by only using the title, the issue's description has been considered as well. The two sets of requirements have slightly different characteristics. In the first phase of the Mylyn project, more complex requirements concerning the basic functionality have been implemented, and in the later phase of the project more requirements concerning small and advanced functionality, respectively.

To identify the code related to the requirements, a specific *VCS* version tag has been used (cf. Section 2.1.2). For each dataset all interaction log entries (i.e. recorded interaction events) have been sorted in chronological order, and then the first version tag after the last interaction log entry has been used. The assumption for this selection procedure is that the *VCS* version which was selected in this way comprises the implementation of the 50 requirements. From these implementation artefacts all artefacts which are not textual and cannot be processed with *IR*, such as pictures or binaries, have been removed (cf. Section 7.2.1).

Table 7.1. Overview of the used Mylyn Project Study Datasets

Dataset	#Requirement	#Int. Log Entries	VCS Version Tag	#Impl. Artifacts		
				All	Textual	Touched by Int.
M_{2007}	50	7687	*R_2_0_RC1*	1103	756	585
M_{2012}	50	1660	*R_3_8_3*	3451	2119	172

Table 7.1 shows the overview of both datasets. As expected, there are much more implementation artefacts in the second (later) M_{2012} dataset than in the first M_{2007} dataset. In contrast, the amount of interaction log entries, both overall and also for each requirement, is lower in the second dataset M_{2012} than in the first M_{2007} dataset. Therefore, only a minor part of all implementation artefacts is directly touched by interactions.

7.2.3 Student Internship 2017

In the following paragraphs the processing of the data to create the S_{2017} data is described, on the basis of the selection of usable requirements, the corresponding recorded interactions, and finally the used source code files.

First out of the overall 42 user stories in the project, 21 user stories assigned to the 3 developers for whom usable interaction recording existed were selected. Overall, the interaction logs of these three developers contained more than two million log entries. The developers recorded these interactions while working on the 21 distinct user stories and the corresponding 98 sub-tasks and touching 312 distinct source code files.

Furthermore, two of the 21 user stories and the corresponding interaction recording were excluded. For one user story one developer did not stop the interaction recording, and thus links to almost all source code files in the Git *VCS* repository were created. The other user story was the first in the interaction logs of a developer and no activation event was recorded for that user story. After also applying the general interaction and source code alignment steps (cf. Section 7.2.1) the S_{2017} dataset contains 19 stories and 98 sub tasks as requirements, 91 JavaScript source code files and 1.171.290 interactions.

7.2.4 Student Internship 2018

In the following paragraphs the processing of the data from the three different data sources to create the S_{2018} data is described.

Source Code in the Git Version Control System

The Git *VCS* repository of the project comprises 406 commits, of which 226 commits (55.67% of all commits) contained a Jira issue ID, which is a similar proportion to that which is reported by others [Rath et al., 2017].

The first 395 commits in the Git repository were used for link creation. The 395th commit is the commit for the completion of the project's last sprint. Commits after the 395th commit did not contain issue IDs and were performed to refactor the source code to the customer's needs after the final project presentation. The Git repository for the 395th commit contained 40 Java and 26 XML files.

Requirements as Issues in Jira

After the project was completed, there were 23 story issues in the Jira project. However, three of the story issues did not specify requirements, but instead testing and project organization. Therefore, these three user stories were removed from the evaluation. Furthermore, the processing status of three story issues was unresolved at the end of the project, and in addition all sub-tasks of these three unresolved stories were unresolved as well. Therefore, these three stories and their interaction recordings were also removed from the evaluation. Finally, the 17 remaining user stories and their 74 sub-tasks along with their interaction recordings were used.

In the project the requirements were specified in German, but the source code files were in English. Thus, the requirements were automatically translated using the *Googletrans*[9] Python library within the evaluation tool (cf. Section 7.4) before preprocessing and *IR*-based link creation. Furthermore, the automatically translated text was also checked randomly for plausibility.

Interaction Recordings

The interaction recordings for the 17 stories and 74 sub-tasks comprise 6471 interaction events separated into 205 commits. After removing interaction events whose files were out of scope (cf. Section 7.2.1), 4012 interaction events were left in the interaction recordings and used for link creation.

Finally, the S_{2018} dataset contains 17 stories and 74 sub tasks as requirements, 66 source code files (40 Java and 26 XML files) and 4012 interactions.

7.3 Gold Standard Creation

Because of the large amount of requirements and source code files in the Mylyn project data sets M_{2007} and M_{2012}, the gold standard creation was based on the correct links created by *ILog* and *IR*. All links for all approaches which were vetted as correct by the study researchers were used as a partial gold standard, and thus only relative recall measures were calculated. For the precision calculation this is not relevant, since only the incorrect links created by an approach influence the precision (cf. Section 2.3.2).

[9] `https://pypi.org/project/googletrans/`, Googletrans documentation website

Table 7.2. Gold Standard Link Candidate Vetting for S_{2017} and S_{2018}

Project		#Stories	#Src Files	#Link Cand.	#Vetted		
					Correct	Wrong	Unknown
S_{2017}	$Dev1_{17}$	3	63	139	37	90	2
	$Dev2_{17}$	11	91	374	128	241	5
	$Dev3_{17}$	5	91	189	52	123	14
	$\sum_{i=1}^{3} Devi_{17}$	19	91	692	**217**	454	21
S_{2018}	$Dev1_{18}$	5	66	330	137	193	0
	$Dev2_{18}$	7	66	462	87	375	0
	$Dev3_{18}$	1	66	66	19	47	0
	$Dev4_{18}$	6	66	396	74	322	0
	$Dev5_{18}$	4	66	264	69	169	26
	$Dev6_{18}$	3	66	198	35	163	0
	$\sum_{i=1}^{6} Devi_{18}$	17	66	1716	**309**	923	26

For the *S2017* and *S2018* projects, a complete gold standard was created by the developers in a similar manner. For the *S2017* project, the gold standard creation was performed in March and April 2017 by the three developers of the project, for whom the interaction log recording was usable for the study, after the project was completed. The upper part of Table 7.2 shows the overview of the link candidate vetting and the resulting gold standard for the *S2017* project. For the gold standard creation, 19 stories of the total number of 42 stories in the Jira project were selected, since these 19 stories were assigned directly to the three developers and described requirements (cf. Section 7.2.3). This limitation ensured that the developers knew the requirements very well.

To limit the link candidates being rated by the developers to a reasonable amount, all possible link candidates between stories and code files tagged with the same feature were considered. For the remaining 19 user stories, all code files from the Git *VCS* repository with the same feature tag (cf. Section 7.1.2) were selected. This excluded particular files which had a format which was different from JavaScript and JSON. Examples of such files are HTML files and build scripts. After this stage, 91 code files, as shown in Table 7.2 remained. As a next step all possible link candidates between stories and code files with the same tag were created. This resulted in 692 link candidates.

For the actual link vetting process a interactive questionnaire spreadsheet with link candidates for each of the three developers was provided. The developers labelled the links as correct (217), wrong (454) or unknown (21), respectively. The latter means that they did not have the competence to judge. The developers also confirmed that all feature labels were correct. The three developers worked on their personalized questionnaire in individual sessions lasting between two to three hours in a separate office room in the research group's department, and there they had the opportunity to ask questions if something was unclear. Thus initially all links of the gold standard were only rated by one developer. During the evaluation study the

link ratings of the developers were checked for plausibility, by inspecting the source code files and requirements involved in each link. In doing so, 113 wrong created links were checked and confirmed by the study researchers.

The gold standard creation for the S_{2018} project was performed in March 2018 by the six developers of the project between the completion of the last sprint and the final presentation to the customer. The lower part of Table 7.2 shows the overview of the link candidate vetting and the resulting gold standard for the S_{2018} project. The developers vetted link candidates between requirements and the source code files in the then actual version in the project's Git *VCS* repository.

The developers vetted the links based on their involvement in the sub-tasks of stories. If there were two developers with an equal number of sub-tasks, both vetted the links and only the links which were vetted as correct by both parties were used in the gold standard. For each developer, a developer-specific interactive questionnaire spreadsheet with all link candidates to vet was generated. This contained all possible link candidates for each user story to all 66 source code files. The vetting resulted in 309 gold standard trace links, where each requirement and each code file was linked at least once.

7.4 Evaluation Tool Support

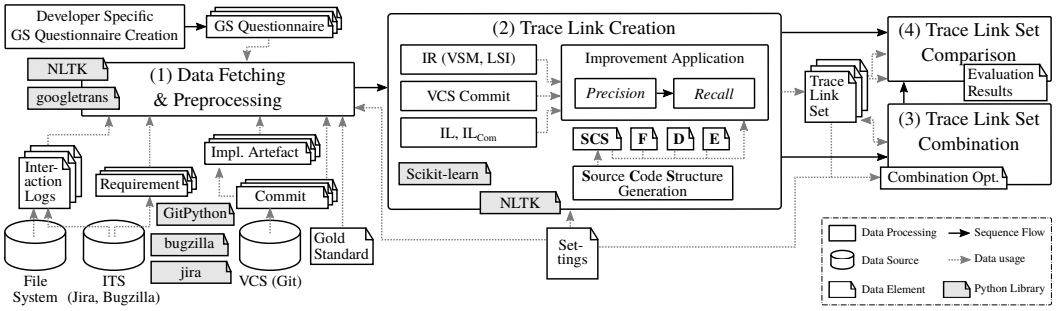

Figure 7.2. ILog Evaluation Tool Suite Data Processing Pipeline

To perform the *ILog* evaluation studies, an evaluation tool suite has been designed and implemented. As shown in Figure 7.2 the evaluation tool suite is implemented as a data processing pipeline. The pipeline comprises steps for different trace link creation methods including improvement techniques, the option to combine multiple set of trace links in different manners (union, intersection, etc.) and the possibility to compare two set of trace links with each other, including the calculation of different evaluation measures (cf. Section 2.3.2). As shown in Figure 7.2 the different trace link creation methods use different data sources, in which *IR*-based techniques use the textual content of artefacts, commit-based link creation uses commit data from an *VCS* repository, and *ILog* uses interaction logs.

The evaluation tool suite is implemented in Python using the *NLTK* (Natural Language Toolkit) and *Scikit-learn*[10] as core libraries. NLTK is used to implement various preprocessing techniques for textual artefacts, such as requirements and source code. For the S_{2018} project, the functionality to automatically translate requirements specified in German into English was implemented using the *Googletrans* Python library. This was necessary since *IR* techniques only work in one language and naturally the source code files of the S_{2018} project were in English. Thus, the tool suite also comprises a preprocessing option for automatically translating text from German into English.

Scikit-learn is used to implement the two *IR* vector space model-based trace link creation techniques, *VSM* and *LSI* (cf. Section 2.2.2.3). Furthermore, the commit-based trace link creation (cf. Section 2.2.3) has been implemented for the Git *VCS* with the *GitPython*[11] library. The *IR*- and commit-based link creation support different setting options. For *IR*-based trace link creation, this is the similarity threshold, and for commit-based link creation there are options as how to handle commits without an issue ID (ignore or assign to next commit with issue ID). In addition, the tool suite also supports the developer-specific automatic creation of questionnaire spreadsheets in order to vet link candidates for gold standard creation.

7.5 Trace Link Creation Techniques

In the following, the notations for the different link creation methods *ILog*, *IR*, and *VCS Commit-based Trace Link Creation* (*ComL*) (cf. Section 2.2.3) used in the *ILog* evaluation studies are introduced. In this, a specific instance of a link creation method is called a link creation technique. *SCS*-based improvement techniques for precision and recall which have been developed for *ILog* (cf. Section 5.2.3) can be applied for *IR* and *ComL* as well, since they only require source code files for their application. Thus, the application of improvement techniques for any link creation method is indicated by a subscript *'i'*. The following listing provides the complete overview of the link creation techniques used:

- ***IL*** denotes the *ILog* approach with manual interaction log assignment (cf. Section 5.2.1.1) and using recorded interactions for link creation. $\boldsymbol{IL_i}$ denotes that interaction data-specific and *SCS*-based improvement techniques have also been applied (cf. Section 5.2.3).
- $\boldsymbol{IL_{Com}}$ denotes the *ILog* approach with commit-based interaction log assignment (cf. Section 5.2.1.2) for link creation by using the recorded interactions and the issue IDs from *VCS* commit messages. $\boldsymbol{IL_{Com\_i}}$ denotes that interaction data-specific and *SCS*-based improvement techniques have also been applied (cf. Section 5.2.3).

[10] `https://scikit-learn.org/`, Scikit-learn documentation website
[11] `https://gitpython.readthedocs.io/`, GitPython documentation website

IR denotes *IR*-based link creation by applying the *VSM* or *LSI IR* technique, while ***IR$_i$*** denotes that *SCS*-based improvement techniques have also been applied (cf. Section 5.2.3). Furthermore, the specific *IR* technique (*VSM* or *LSI*) which was used is denoted as: *<IR technique(similarity threshold)>* with the similarity threshold (cf. Section 2.2.2) used, e.g. *VSM(0.2)* for usage of *VSM* with similarity threshold of *0.2*.

ComL denotes commit-based link creation by using the issue IDs from *VCS* commit messages and the files contained in the commits. ***ComL$_i$*** denotes that *SCS*-based improvement techniques have also been applied (cf. Section 5.2.3).

7.6 Proceeding of Evaluation Studies and their Characteristics

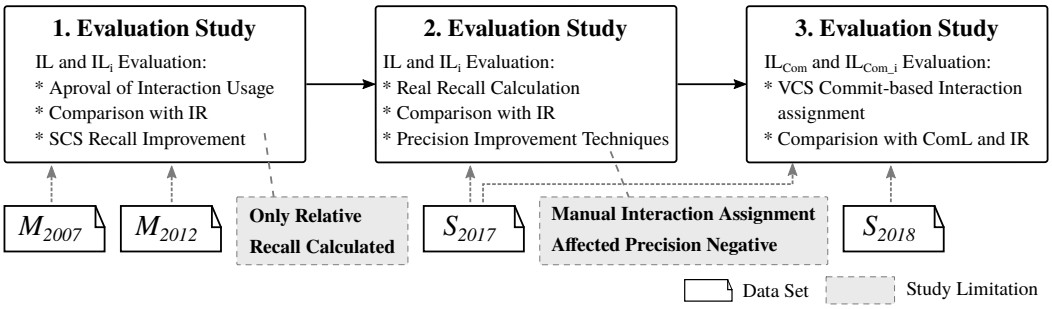

Figure 7.3. ILog Evaluation Studies Overview, Proceeding and Dataset Usage

Figure 7.3 shows the overview of the proceeding for the three performed *ILog* evaluation studies, along with the datasets used and the limitations of the studies that lead to the final version of *ILog*. The intention of the first evaluation study was to approve that interaction log recordings are a reasonable data source for trace link creation, and thus confirm the basic idea of the *ILog* approach. The previously introduced M_{2007} and M_{2012} datasets from the Mylyn project were used in the study, while the *ILog* links were generated by the *ILog with manual performed interaction assignment* (*IL*) technique and compared to *IR* created links. Furthermore, *SCS*-based recall improvement was applied for both link creation techniques. However, due to the size of the M_{2007} and M_{2012} datasets, and specifically due to the huge overall number of possible links and the resulting huge effort to create a complete gold standard for the datasets, only relative recall was calculated, which initially motivated the second study.

In the second study the S_{2017} data set was used. Initially trace links were created, as in the first study. However, since the precision of links created with *IL* was not satisfying for directly using the links, wrong link detection techniques to improve the

precision were developed (cf. Section 5.2.3.1). Applying the precision improvement techniques, through the use of IL_i instead of IL, improved the created links but the links were still not good enough to be directly usable.

After further analysis of the S_{2017} dataset, it turned out that the manual assignment of interaction recordings was not used as had been intended by the student developers of the project. This led to the development of *ILog with VCS Commit-based interaction assignment* (IL_{Com}) (cf. Section 5.2.1.2), initially using the S_{2017} dataset and simulating the assignment of interaction recordings by using issue IDs from commit messages instead of a manual assignment.

The positive results of the simulated IL_{Com} application with the S_{2017} dataset, motivated the evaluation of IL_{Com}, in a project directly using commit-based interaction assignment in the third evaluation study. Thus, the S_{2018} dataset was created by directly applying IL_{Com} during the project. In addition to the former studies' application of improvement techniques and their comparison with other link creation techniques, trace links were also created by only using commit data, in the form of *ComL* (cf. Section 2.2.3).

Table 7.3. ILog Evaluation Studies Characteristics and Dataset Usage

		1. Eval. Study		*2. Eval. Study*	*3. Eval. Study*
Characteristic		M_{2007}	M_{2012}	S_{2017}	S_{2018}
Study & Project Basics	Eval. intent chpt.	initial *ILog* *Chapter 8*		improvement tech. *Chapter 9*	IL_{Com} commit assig. *Chapter 10*
	Organisation	Open Source		Scrum	Scrum
	# developers	19 (3)	20 (3)	7 (3)	6
Require-ments	*ITS*	Bugzilla		Jira	Jira
	Format	Feature		Story (Sub task)	Story (Sub task)
	#	50	50	19 (98)	17 (74)
Source Code	*VCS*	Git/ SVN		Git	Git
	# commits	n/a		761	406
	# files	756	2119	91	66
	Format	Java, XML		JavaScript	Java, XML
Interaction Recording	*IDE*	Eclipse		IntelliJ/ Webstorm	Eclipse/ ADT
	Asg-mt. tool	Mylyn		*ILog* Activity Tracker	*ILog* Eclipse Client
	type	manual		manual	commit
	#	7687	1660	1171290	4012
Trace Link Creation	*IR*	*VSM*, *LSI*		*VSM*, *LSI*	*VSM*, *LSI*
	ComL	n/a		Commit + issue id	Commit + issue id
	ILog	*IL*		*IL*, IL_{Com}	IL_{Com}
	Imp-rove. data	*SCS*		*SCS*, inter. data	*SCS*, inter. data
	type	R		P, R	P, R
	Eval. meas.	$P, R_r; f_1, f_{0.5}$		$P, R; f_1, f_{0.5}$	$P, R; f_1, f_{0.5}$

Table 7.3 shows more detailed characteristics of the three evaluation studies along with the datasets used. In the first initial evaluation study, with the M_{2007} and M_{2012} datasets, *IL* has been evaluated regarding precision and relative recall of the resulting trace links. For comparison, trace links are also created with the two *IR* techniques *VSM* and *LSI*. In addition to precision and relative recall $f0.5$ and f_1, measures are calculated for the overall rating of the evaluation results.

In the second evaluation study with the S_{2017} dataset the different precision and recall improvement techniques using *SCS* and interaction data were evaluated (cf. Section 5.2.3). Despite the different requirements and source code formats, the biggest difference to the M_{2007} and M_{2012} datasets is the number of recorded interactions. Over one million interactions were recorded for the S_{2017} dataset. The reason for this large number is the different interaction recording tool and *IDE*. In IntelliJ the *IDE's APIs* used by the *Activity Tracker*-based interaction recording tool creates very detailed interaction events for all interactions performed, such as for every key stroke an interaction event is generated. In the Eclipse *IDE* and Mylyn the interaction events which are generated and used are on a higher level and focused on source code-related interactions, such as adding a new method to a class, which would only result in two recorded interaction events. A further difference to the first study is the calculation of real recall instead of relative recall, which was possible since the student developers created a complete gold standard for the links.

In the third evaluation study with the S_{2018} dataset, IL_{Com} with a commit-based assignment of recorded interactions to requirements was evaluated (cf. Section 5.2.1.2). This marks the biggest difference compared to the other two studies, in which *IL* with its manual interaction assignment was used instead. Furthermore, *ComL*, i.e. trace link creation which only relies on commit data (cf. Section 2.2.3), is added as another link creation technique for comparison. Before performing the study with the S_{2018} dataset, the S_{2017} dataset has been used to retrospectively simulate the application of IL_{Com} and also to create links with *ComL*. This was possible since the students followed the convention of providing issue IDs in the commit messages in the S_{2017} project as well. The positive results of the retrospective application of IL_{Com} with the S_{2017} dataset was the initial motivation to conduct the third study with the S_{2018} dataset.

Since the user stories of both student projects S_{2017} and S_{2018} contained only short texts, the threshold values used for *IR* (cf. Section 2.2.2) had to be set low. Furthermore, *SCS*-based precision and recall improvements (cf. Section 5.2.3.2) have also been applied to the links created with *IR* and *ComL* in both projects.

Chapter 8

Using Interaction Logs for Trace Link Creation

In this chapter the initial evaluation study for the *ILog* approach is described. After in the previously conducted problem investigation, it was shown that existing trace link creation techniques are insufficient for the continuous creation of directly usable links. The *ILog* approach has been designed and introduced to treat this problem. Thus, this first evaluation study is the start of the treatment validation task in the design cycle of the thesis. The goal of this initial evaluation study is to show that the usage of interaction log data for trace link creation is reasonable. This goal is achieved by answering the following general research questions: *What is the precision and recall of ILog's link creation technique IL?*, and *How do precision and recall values of IL compare to IR for the same dataset?*

Section 8.1 introduces the experimental design of the study and consists of the detailed research questions, an overview of the experimental activities performed, how trace links have been created with the different trace link creation techniques using the previously introduced M_{2007} and M_{2012} datasets, and how the trace links resulting from the different creation techniques have been evaluated. Section 8.2 presents the results by answering the previously stated research questions. Section 8.3 summarizes and discusses the results of the study, which showed that the *IL* technique of the *ILog* approach achieves very good precision and good recall with the interaction log recording data from the Mylyn open source project, and outperforms different *IR*-based link creation techniques.

8.1 Experiment Design

In this section the details of the study's experimental design, as shown in Figure 8.1, are described, in particular wrt. usage of the M_{2007} and M_{2012} Mylyn project datasets and the application of *IL*.

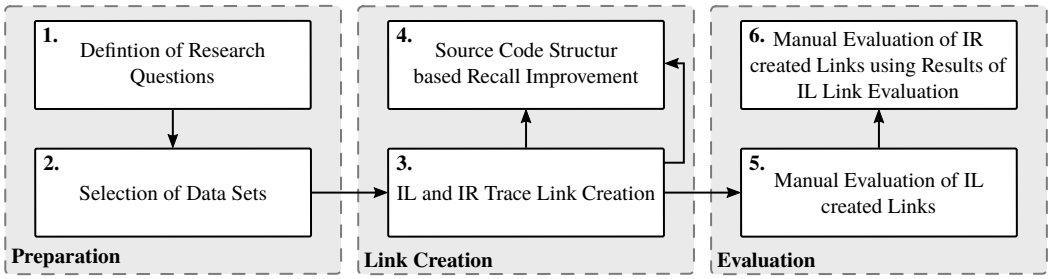

FIGURE 8.1. 1. STUDY EXPERIMENTAL DESIGN: OVERVIEW OF ACTIVITIES PERFORMED

8.1.1 Research Questions

The study's overall research question – *Is there a difference between the application of IL and IR based trace link creation regarding precision and relative recall?* – is divided into three sub-questions:

RQ1: *What is the precision of IL and IR-created trace links?* The hypothesis was that the precision of *IL* is better than *IR*, since *IL* link creation is based on developers' expert knowledge.

RQ2: *What is the relative recall of IL and IR-created trace links?* The hypothesis was that the relative recall of *IL* is at least as good as the relative recall of *IR*. On the one hand, *IL* can find links between artefacts which are not textually similar. On the other hand, artefacts found by *IR* are most likely also covered by interactions.

RQ3: *What is the impact of using Source Code Structure (SCS)?* The hypothesis was that using the *SCS* in addition to *IL* and *IR* improves the relative recall of the created trace links.

As shown in Figure 8.1, the experimental design of the study is separated into three parts in order to answer these research questions:

Preparation The experimental design is guided by the previously stated detailed research questions. In the experiment the two datasets M_{2007} and M_{2012}, both of which are taken from the Mylyn project (cf. Section 7.1.1) are used. For each of the two datasets the experimental steps are:

Link Creation Creation of trace links with *ILog's IL* technique and the two *IR* techniques *VSM* and *LSI* (cf. Section 8.1.2), and application of *SCS* recall improvement (cf. Section 5.2.3.2) for links created with all techniques.

Evaluation Manual evaluation of the trace links created with *IL* followed by manual evaluation of the trace links created with *VSM* and *LSI*. In the evaluation of *VSM* created trace links, the links which are already verified in the manual evaluation of *IL* trace links were used, and in the manual evaluation of *LSI* created trace links the links which were already verified from *IL* and *VSM* were used.

8.1.2 Trace Link Creation

IL links have been created using the interaction log recordings of the M_{2007} and M_{2012} datasets. For *IR*-based trace link creation, both *IR* techniques *VSM* and *LSI* have been applied to the two datasets M_{2007} and M_{2012}. The preprocessing steps as described in Section 2.2.2.1 have been applied upfront to all the artefacts used. Furthermore, the trace link candidate generation has been restricted to links from requirements to implementation artefacts (source code files), which is to say that no trace links between the same artefact types were created.

Table 8.1. Thresholds and Number of Candidate Links for IR Techniques

Thresholds*		0.9	0.8	0.7	0.6	0.5	0.4	0.3
M_{2007}	*VSM*	0	50	596	2347	6419	13798	24040
	LSI	0	3	8	40	142	354	1058
M_{2012}	*VSM*	185	2268	6431	12333	22397	39434	64284
	LSI	1	14	86	297	920	2424	6014

* Selected values are highlighted

To determine a reasonable threshold for the *IR* techniques with the initially approved threshold values of 0.7 for *VSM* [Lucia et al., 2004] and 0.3 for *LSI* [De Lucia et al., 2007] have been used. While this worked well for M_{2007}, a different thresholds for M_{2012} had to be chosen. As can be seen from Table 8.1, which shows the number of link candidates for different *IR* techniques and thresholds, the number of generated links for the second M_{2012} dataset increases very quickly in lowering the threshold. To limit the effort required for the verification of the link candidates, thresholds which resulted in less than 1000 links have been used. Clearly, the results for M_{2012} can only be seen as a first indication and they might yet be improved in future with lower thresholds.

8.1.3 Data Evaluation

To evaluate the trace links created with the *IL* technique, the created links from both datasets M_{2007} and M_{2012} have been compared with links created by *VSM* and *LSI*. Two settings for trace link creation have been used: one with *SCS* recall improvement, and one without. The following steps to determine *True Positive* (*TP*) (correct) and *False Positive* (*FP*) (wrong) links for these link sets have been performed. Initially, the links created by *IL* have been verified manually for the M_{2007} and M_{2012} dataset. Subsequently, these verified links have been removed from the *VSM* and *LSI* trace link candidate sets. This resulted in sets of link candidates which were found only by the two *IR* techniques. These links have also been manually verified. Finally, the verified *VSM* and *LSI* links and the verified *IL* links have been used to determine the set of links which were found only by *IL*.

8.2 Results

In the following subsections, the research questions of this study are answered and the results are discussed. In Section 8.2.1 *RQ1* and *RQ2* concerning precision and relative recall of *IL* and *IR* created trace links is answered. This is followed by the answer to *RQ3* concerning the *SCS* based recall improvements in Section 8.2.2.

8.2.1 Evaluation of IL and IR based Trace Link Creation

Table 8.2. Comparison of IR and IL Trace Link Creation

	M_{2007}			M_{2012}		
	IL	*IR VSM*(0.7)	*IR LSI*(0.3)	*IL*	*IR VSM*(0.9)	*IR LSI*(0.5)
#Link Cand. (LC)	1148	596	1058	240	185	920
#Impl. $\text{Artefact}_{\text{LC}}$	585	203	384	172	171	444
$\#\text{Requirements}_{\text{LC}}$	50	23	46	37	4	34
# *True Positive* (*TP*) Trace Links	1148	204 ($118_{IL} + 17_{LSI} + 69$)	328 ($184_{IL} + 37_{VSM} + 107$)	240	25 ($6_{IL} + 24_{LSI} + 1$)	274 ($41_{IL} + 24_{VSM} + 250$)
#Impl. $\text{Artifact}_{\text{TP}}$	585	126	200	240	24	169
$\#\text{Requirements}_{\text{TP}}$	50	19	41	172	3	28
# *TP* Trace Links of all Approaches		1324 ($1148_{IL} + 69_{VSM} + 107_{LSI}$)			491 ($240_{IL} + 1_{VSM}250_{LSI}$)	
Precision	1	0.341	0.310	1	0.135	0.298
Relative Recall	0.867	0.154	0.247	0.418	0.051	0.534

Table 8.2 provides an overview of the number of created trace link candidates, implementation artefacts used, requirements used, correct (*TP*) trace links, implementation artefacts involved in correct (*TP*) trace links, requirements involved in correct (*TP*) trace links, sum of correct trace links created by all approaches together, and the precision and relative recall for both datasets. Thus, the research questions can be answered as follows.

RQ1: What is the precision of IL and IR-created trace links? For both datasets all links created with *IL* were correct (100% precision). For *IR* precision values vary between 13% and 34% with little difference between *VSM* and *LSI* for the standard thresholds in the first M_{2007} dataset and a big difference for the higher thresholds in the second M_{2012} dataset. Thus, *IL* clearly outperforms *IR*. Moreover, *IL* is independent of setting a threshold and finds more correct links than *IR* for the M_{2007} dataset. Nevertheless, there are also links which are only discovered by *IR* in this dataset.

For the M_{2012} dataset the situation is different due to the smaller number of *IL* created trace links and due to the much larger amount of implementation artefacts used for *IR*. Beyond that, not all requirements are involved in interaction links in this set. This is due to the fact that some interactions concerned code which lay outside of the *VCS* tag (e.g. the framework used). *LSI* finds in total more correct trace links for the second dataset than *IL*. This can be explained by the amount of considered requirements and implementation artefacts, since *IL* considered 37 requirements and 172 implementation artefacts, whereas *LSI* considered 34 and 444. In comparison with the values achieved by trace link creation approaches and

techniques, as discussed in Section 3.2 of Chapter 3, it can be stated that the 100% precision of *IL* in a real-world setup is unique. The precision of *IR* is acceptable for the first M_{2007} and good for the second M_{2012} dataset. The values for precision are in the range reported by De Lucia et al. [2007] (*LSI*), Ali et al. [2013] (*VSM*) and Merten et al. [2016b] (*LSI*, *VSM* and *ITS* as data source).

RQ2: What is the relative recall of IL and IR-created trace links? The setting used in the experiment resulted in relative recall values between 86% and almost 42% for *IL* (cf. Table 8.2) and relative recall values of 5% and 53% for *IR*. As was expected and reported by others Cleland-Huang et al. [2007], Gotel et al. [2012b], *IR* creates a lot of *False Positive* (*FP*) trace links even with the moderate threshold setting as was used for the second M_{2012} dataset in the experiment. The difference in relative recall rates between the M_{2007} and M_{2012} datasets in *IL* can be explained by the characteristics of the datasets which resulted in a lower number of interactions for the second M_{2012} dataset (cf. Section 7.2.2, Table 7.1: M_{2007} has 7687 interactions on 756 used implementation artefacts, M_{2012} has 1660 interactions on 2119 used implementation artefacts).

8.2.2 Source Code Structure based Recall Improvement

During the study it turned out that the recall of *IL* was bad for specific requirement issues compared to *IR*. This was based on the fact that only a small amount of interactions existed, since the developer implementing the specific requirement seemed to have very detailed knowledge about the code. As a result, implementation artefacts which were indirectly related and which were also important for the requirements implementation were not touched. To countervail this, the *SCS*-based recall improvement was developed (cf. Section 5.2.3.2).

Table 8.3. Trace Links for different Code Traversal Levels

Traversal Level	$M_{2007}IL$			$M_{2007}IR$					
	#Link Cand.	#*TP* Links	Precision	#Link Cand.		#*TP* Links*		Precision	
				VSM(0.7)	*LSI*(0.3)	*VSM*(0.7)	*LSI*(0.3)	*VSM*(0.7)	*LSI*(0.3)
0	1148	1148	1.000	596	1058	120	184	0.201	0.174
1	1446	1446	1.000	858	1718	234	338	0.273	0.197
2	1831	1831	1.000	1108	2181	363	562	0.328	0.258
3	2204	2204	1.000	1382	2706	499	805	0.361	0.297
4	2565	2565	1.000	1624	3214	639	1083	0.393	0.337
5	3027	2854	0.943	1915	3927	781	1349	0.408	0.344
6	3531	3202	0.907	2253	4510	947	1612	0.420	0.357
10	5805	3639	0.627	3374	5488	1258	1779	0.373	0.324

* Compared to *IL*

As introduced in Section 5.2.3.2, the setting of an appropriate traversal level is important to apply the *SCS* based recall improvement without negative effects on the precision. Table 8.3 shows the differences according to the number of created links and their precision for considering *SCS* with different traversal levels for the M_{2007} datasets. $M_{2007}IL$ refers to links generated by *IL* and $M_{2007}IR$ refers to links

generated by *IR*. Since precision for *IL* drops when considering a traversal level of code relations greater than four, this traversal level was used for the answer of *RQ3*. It can also be seen that precision of *IR* only drops for traversal level ten. As the focus was to maximize the precision of *ILog*, level four was chosen. Clearly, the results for *IR* could be improved with a higher traversal level. This analysis was also performed for the second M_{2012} dataset. Since the results were quite similar, their detailed reports are skipped.

Table 8.4. IR and IL Trace Links Considering Source Code Structure

	M_{2007}			M_{2012}		
	IL_i	IR_i *VSM*(0.7)	IR_i *LSI*(0.3)	IL_i	IR_i *VSM*(0.9)	IR_i *LSI*(0.5)
#Link Cand. (LC)	2565	1624	3143	1126	458	2766
#Impl. $\text{Artifact}_{\text{LC}}$	627	333	516	363	343	702
#$\text{Requirements}_{\text{LC}}$	50	23	46	37	4	34
#*True Positive* (*TP*) Trace Links	2565	698 $(581_{IL} + 63_{LSI} + 54)$	1214 $(1010_{IL} + 62_{VSM} + 142)$	1126	108 $(91_{IL} + 17_{LSI} + 0)$	784 $(491_{IL} + 11_{VSM} + 282)$
#Impl. $\text{Artifact}_{\text{TP}}$	627	229	308	363	73	354
#Requirements (TP)	50	22	41	37	4	35
#$\text{Trace Links}_{\text{TP}}$ of all Approaches		2761 $(2565_{IL} + 54_{VSM} + 142_{LSI})$			1408 $(1126_{IL} + 0_{VSM} + 282_{LSI})$	
Precision	1	0.425	0.386	1	0.236	0.283
Relative Recall	0.929	0.253	0.440	0.800	0.077	0.557

RQ3: What is the impact of using Source Code Structure (SCS)? As shown in Table 8.4 for both datasets, all links created with *IL* were also correct (100% precision) when considering *SCS*. Furthermore, relative recall increased considerably for the second M_{2012} dataset. Comparing the numbers in Table 8.2 and 8.4 it can be seen that the *SCS*-based recall improvement for *IL* results in five times more trace links for the second M_{2012} dataset and twice as many links for the first M_{2007} dataset. This can be explained by the more complex *SCS* due to the maturity of the project in the second dataset and by the larger amount of links which are already created by the larger amount of interactions.

Both *IL* and *IR* considered about 1/3 more implementation artefacts when using *SCS*. For *VSM* and *LSI* in the M_{2007} dataset, this resulted in an increase of precision and relative recall. This is also true for the M_{2012} dataset, except for the precision value of *LSI*, which drops slightly.

In the experiments with different code traversal levels, the number of links only found by *IR* could be reduced to almost zero by increasing the traversal levels of code relations. However, this also resulted in false positive links for *IL*, which is contrary to the research goal **G1**, namely to create trace links with 100% precision in order to make them directly usable. Altogether, it can be seen that by using *SCS* to improve recall, the research goal of 100% precision and excellent relative recall was achieved and that *IL* outperforms *IR* for both Mylyn project datasets.

8.3 Conclusion

The results for ILog's *IL* technique of this first study are encouraging. *IL* created trace links with 100% precision for two different datasets. Also, the calculated relative recall values are excellent, since they are almost 96% for the first M_{2007} dataset and 80% for the second M_{2012} dataset. Thus, the study shows that, with ILog's *IL* technique, trace link creation in practice can be supported with little extra effort for the developers. Clearly, the comparison with *IR* is only preliminary in this study, since specific thresholds for the second dataset were used and only relative recall was computed.

Chapter 9

Improvement Techniques for Interaction Log Trace Links

In this chapter the second evaluation study of the *ILog* approach is presented. In contrast with the first study, this study is based on interaction log data, requirements and source code from the student project S_{2017} (cf. Section 7.1.2). A student project was used in order to be able to create a complete gold standard, with the help of the student developers. Also in contrast with the first evaluation study, the complete gold standard enabled the calculation of real recall instead of only relative recall.

The study consists of two parts. In the first part precision and real recall values for the *ILog's IL* technique were calculated. The first results of the study showed that *IL* only had around 50% precision when applying it in the same manner as in the first previous study. It turned out that the bad precision and thus wrong links were caused by developers not triggering the interaction recording for requirements correctly. Since *IL* was used in the study, the interactions captured were assigned manually to requirements by a manually performed indication within the *IDE* of the developers. However, the developers worked on different requirements without changing the requirement in the *IDE*. Thus, all trace links were created for one requirement.

As a result of this, in the second part of the study the precision improvement techniques for detection of potentially wrong links (cf. Section 5.2.3.1) were developed and evaluated. With these techniques, the precision is improved by identifying relevant trace link candidates, such as a focus on links created by edit interactions or thresholds for the frequency and duration of interactions. Furthermore, different techniques to identify irrelevant source code, such as the developer who created the source code, or source code which does not refer to other source code in interaction created trace links, were both evaluated. In the best cases this improved the precision up to almost 70% with still reasonable recall above 45%. Thus, in addition to the general research questions of the previous study concerning the link creation

quality of *ILog's IL* technique, this study answers the general research question: *To what extent can improvement techniques countervail wrong and missing ILog links?*

The chapter is structured similarly to the previous study chapter. Section 9.1 introduces the experimental design of the study, and consists of the detailed research questions, an overview of the experimental activities performed, how trace links have been created initially with the different trace link creation techniques using the previously introduced S_{2017} dataset, how the resulting trace links from the different creation techniques have been evaluated and how the trace link improvement techniques have been applied to the trace links which were initially created from the different techniques. Section 9.2 presents the results by answering the previously stated research questions together with a discussion. Section 9.3 summarizes the results of the study, which have shown that the improvement techniques which were evaluated are suitable to remove wrong links and thus improve the precision of *IL* created links.

9.1 Experiment Design

In this section the details of the study's experimental design, as shown in Figure 9.1, are described, in particular wrt. usage of the S_{2017} student project dataset and the development and application of *ILog's* precision improvement techniques (cf. Section 5.2.3.1).

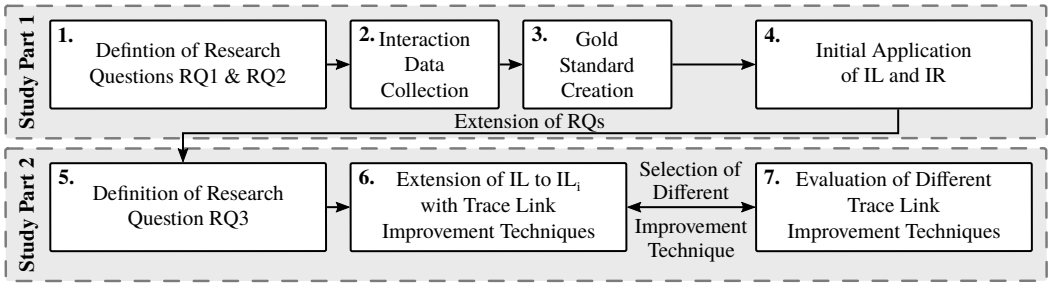

Figure 9.1. 2. Study Experimental Design: Overview of Activities Performed

9.1.1 Research Questions

The research questions answered in the two parts of this study are:

RQ1: *What is the precision and recall of IL created trace links?* The hypothesis was that *IL* has very good precision and good recall.

RQ2: *What is the precision and recall of IR created trace links?* The hypothesis was that *IR* has bad precision and good recall.

RQ3: *What is the precision and recall of IL_i when precision improvement techniques for the detection of wrong trace links are applied?* The hypothesis was that improvement techniques utilizing details of the interaction log, such as the frequency, and improvement techniques which consider the source code, such as by using the *SCS*, should enhance precision considerably and still keep reasonable recall.

As shown in Figure 9.1, *RQ1* and *RQ2* were defined for the first part of the study and targeted the calculation of real recall values instead of relative recall, as in the previous study for *IL* (*RQ1*) and additionally for comparison with *IR* (*RQ2*). After the initial application of *IL*, it turned out that the precision of *IL* was not sufficient for direct usage of the trace links with the S_{2017} student project dataset. Thus, in the second part of the study precision, improvement techniques were investigated in *RQ3*.

9.1.2 Part 1: Initial Trace Link Creation

Initially trace links were created with *ILog's IL* technique and with the *VSM* and *LSI IR* techniques. All link creation techniques were applied to the 19 stories, together with their 98 sub-tasks and to the 91 source code files which were also used for the gold standard creation of the S_{2017} dataset (cf. Section 7.6 and the S_{2017} characteristics in Table 7.3).

For *IL* the interactions of a story were combined with the interactions of the corresponding sub-task for further evaluations, as the sub-tasks describe details for implementing the story (cf. Section 5.2.2). From the resulting link candidates all links to code files which were not included in the 91 code files of the gold standard were removed. *IR* was applied to the texts of stories and corresponding sub-tasks and to the 91 code files used for the gold standard. In addition, the common *IR* preprocessing steps, i.e. stop word removal, punctuation character removal and stemming, were performed [Baeza-Yates and Ribeiro, 2011, Borg et al., 2014]. Besides this, the camel case identifier splitting (e.g. as when *PatientForm* becomes *Patient Form*), was performed since camel case notation has been used in the source code (cf. Section 2.2.2.1). Since the stories contained only very short texts, the threshold values used for the *IR* techniques had to be set very low.

9.1.3 Part 2: Precision Improvement Techniques

Since *ILog's IL* technique had worse precision values than expected, it was decided to investigate how *IL* can be extended by improvement techniques for the detection of wrong trace links. To this end, the initial study was extended by means of a second part in which *RQ3* (cf. Section 9.1.1), which concerns the evaluation of improvement techniques to detect wrong links, is answered. Therefore, the improvement

techniques (cf. Section 5.2.3) with different settings for each improvement technique and the most promising combination of improvement techniques were evaluated, as will be described in the following.

For interaction-specific data (a) the type of interaction, i.e. whether an interaction is a select or an edit, (b) the duration of interactions based on the logged time stamp, and (c) the frequency with which an interaction with a source code file occurred for a user story, were evaluated. The rationale for this was that (a) edit events are more likely than select events to identify code which is necessary for a user story and that (b, c) a longer duration of the interaction or a higher frequency signify that the developer made a more comprehensive change and not only a short edit, e.g. by correcting a typo which was noticed when looking at a file.

For source code (a) the ownership that is the developer who created the interaction, since one developer might have worked in a less disciplined way than others; (b) the frequency of interactions with the same source code files for different user stories, as files used for different user stories might be base files which had not been considered relevant for the gold standard by the developers; (c) filtering is performed with only JavaScript source code files, as other formats might not be so relevant for a user story; and (d) the *SCS* for the source code files which are involved in one story to detect files which had no relation in the *SCS* to other files, as the unrelated code files might signify a different purpose from the story. As a consequence of this, the most promising techniques were combined.

Altogether, the improvement techniques to detect wrong links were applied in such a way that links were removed when their logged interaction data values fell below a certain threshold, different with respect to a certain type, or when the source code file did not match the aforementioned criteria. Finally, the thresholds, the type and the combination of thresholds and source code filter criteria used to optimize the precision of the links created by *IL* and to minimize the effect on the recall, were chosen.

9.2 Results

This section reports the results of the performed evaluations and answers the research questions *RQ1* and *RQ2* in Section 9.2.1 and *RQ3* in Section 9.2.2. In addition Section 9.2.3 discusses the results.

9.2.1 Part 1: Precision and Recall for the Initial Evaluation

Table 9.1 presents an overview of the evaluations performed, as described in section 9.1.2. *ILogs's IL* technique created 372 link candidates, of which 212 were wrong. 57 correct links were not found. Thus *RQ1* can be answered as follows: the precision for the *IL* technique is 43.0% and recall is 73.7%.

Table 9.1. Precision and Recall for IL and IR

Approach	*GS* Links	Link Cand.	Correct Links	Wrong Links	Not Found	Precision	Recall	$F_{0.5}$	F_1
IL	217	372	160	212	57	0.430	0.737	0.469	0.543
$IR_{VSM(0.3)}$	217	191	38	153	179	0.199	0.175	0.194	0.186
$IR_{VSM(0.2)}$	217	642	104	538	113	0.162	0.480	0.187	0.242
$IR_{LSI(0.1)}$	217	102	35	67	182	0.343	0.161	0.280	0.219
$IR_{LSI(0.05)}$	217	363	77	286	140	0.212	0.355	0.231	0.266

RQ2 can be answered by looking at different *IR* techniques with different thresholds: the best achievable precision with very low thresholds is 34.3% ($LSI(0.1)$) and the best achievable recall is 48.0% ($VSM(0.2)$). These results are not good at all compared to *IL* and they are bad compared to typical *IR*-results on structured data [Hayes et al., 2006] (cf. Section 2.3.2).

As *IL's* precision was much lower than expected, it was investigated whether there was a problem with the gold standard (cf. Section 7.3). In this connection, 113 wrong links which resulted from edit interactions were checked manually. The performed check confirmed that these links are really wrong. This led to the conclusion that the developers had not used the interaction logging properly and that they had worked on code which was not relevant for the activated user story. This happened typically for smaller code changes on the fly besides the implementation of the activated story. So for example, developers updated a file from which they had copied some code, but they did not activate the requirement that the change should have been associated with.

9.2.2 Part 2: Precision and Recall Using Improvement Techniques

Table 9.2. Duration-based IL Improvement

Dur. (sec)	GS Links	Link Cand.		Correct		Wrong		Not Found	Precision		Recall		$F_{0.5}$	F_1
		All	Edit	All	Edit	All	Edit		All	Edit	All	Edit		
1	217	372	220	160	107	212	113	57	0.430	0.486	0.737	0.493	0.488	0.490
10	217	317	199	144	104	173	95	73	0.454	0.523	0.664	0.479	0.513	0.500
60	217	231	167	113	90	118	77	104	0.489	0.539	0.521	0.415	0.508	0.469
180	217	183	142	93	78	90	64	124	0.508	0.549	0.429	0.359	0.497	0.435
300	217	154	122	81	70	73	52	136	0.526	0.574	0.373	0.323	0.496	0.413

This section reports the answers to *RQ3*. Table 9.2 shows the results which focus on links created by edit interactions and which possess a different minimal duration. The first row corresponds to *IL* without any restrictions. It shows that, by focusing on edit interactions, the precision slightly improves from 43.0% to 48.6%. As the focus on edit always improved the precision a little, only the F-measures for *IL* which were focused on edits are reported and only these numbers are described in the following text. When increasing the minimum duration for an interaction, the precision can be improved up to 57.4%. This of course impairs the recall. However,

as reported at the end of this section *IL's SCS*-based recall improvement (cf. Section 5.2.3.2) can balance the negative effect on recall up to a certain extent.

TABLE 9.3. FREQUENCY-BASED IL IMPROVEMENT

Frequency	*GS* Links	Link Cand.		Correct		Wrong		Not Found	Precision		Recall		$F_{0.5}$	F_1
		All	Edit	All	Edit	All	Edit		All	Edit	All	Edit		
1	217	372	220	160	107	212	113	57	0.430	0.486	0.737	0.493	0.488	0.490
2	217	314	220	142	107	172	113	75	0.452	0.486	0.654	0.493	0.488	0.490
5	217	220	191	113	98	107	93	104	0.514	0.513	0.521	0.452	0.499	0.480
10	217	181	169	99	93	82	76	118	0.547	0.550	0.456	0.429	0.521	0.482
20	217	158	151	90	87	68	64	127	0.570	0.576	0.415	0.401	0.530	0.473
100	217	86	86	59	59	27	27	158	0.686	0.686	0.272	0.272	0.526	0.389

Table 9.3 shows the results of different minimal frequencies for trace links within interaction log recording for one story. Again, row one gives the numbers for the original *IL* application. Here, the improvement is greater in leading to a precision of 68.6% for a frequency of 100. In particular, by this restriction, all select interactions are removed. However, recall is even more impaired.

TABLE 9.4. DEVELOPER-SPECIFIC DIFFERENCES

Developer	*GS* Links	Link Cand.		Correct		Wrong		Not Found	Precision		Recall		$F_{0.5}$	F_1
		All	Edit	All	Edit	All	Edit		All	Edit	All	Edit		
Dev_1	37	41	17	19	6	22	11	18	0.463	0.353	0.514	0.162	0.286	0.222
Dev_2	128	252	155	110	79	142	76	18	0.437	0.510	0.859	0.617	0.528	0.558
Dev_3	52	77	46	30	21	47	25	22	0.390	0.457	0.577	0.404	0.445	0.429

Table 9.4 shows the distribution for the three developers of the evaluation. One can see that the developer Dev_2 was the most active and that Dev_3 contributed more than Dev_1. However, for all three developers, the interactions led to more wrong than correct links. So the precision between them all does not differ much.

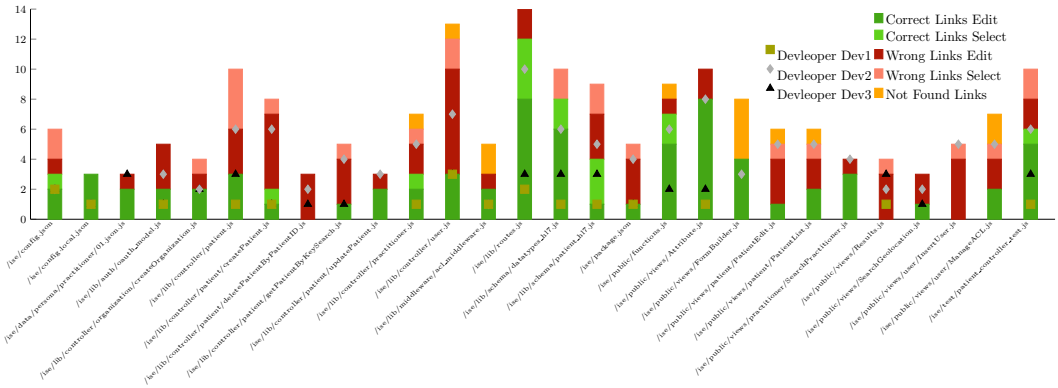

FIGURE 9.2. CODE FILES WHICH HAD INTERACTIONS IN 3 OR MORE USER STORIES

Figure 9.2 shows the 28 code files which have been touched in interactions for three or more user stories. Furthermore, it shows how often each developer touched these files. The developer distribution shows that some of the files have been touched

by different stories from one developer and some have been touched from several developers. One can see that only three out of 28 files have wrong link candidates only. Also, files which have many link candidates sometimes have many correct link candidates and sometimes they do not. So, there is no clear pattern that these files are the reason for the greater number of wrong link candidates.

Table 9.5. Source Code-based Improvements

Code Res.	*GS* Links	Link Cand.		Correct		Wrong		Not Found	Precision		Recall		$F_{0.5}$	F_1
		All	Edit	All	Edit	All	Edit		All	Edit	All	Edit		
none	217	372	220	160	107	212	113	57	0.430	0.486	0.737	0.493	0.488	0.490
>3 US	217	208	92	83	43	125	49	134	0.399	0.467	0.382	0.198	0.368	0.278
Only js	186	327	203	129	99	198	104	57	0.394	0.488	0.694	0.532	0.496	0.509
Con.	217	274	169	147	99	127	70	70	0.536	0.586	0.677	0.456	0.554	0.513

This is confirmed in Table 9.5 which shows the results for the different source code restrictions, with the first row showing the numbers without restrictions. The second row shows the precision for code which was touched by interactions in three or more user stories. Here, the precision increased slightly to 46.7%. The third row shows a precision of 48.8% when looking only at JavaScript files. The best precision of 58.6% could be achieved when removing code files which were not connected by *SCS* relations to other code files of the same user story (cf. Section 5.2.3.1).

When looking at the individual improvement techniques for detecting wrong links, *RQ3* can be answered as follows. The best precision of 68.6% can be achieved with a minimum frequency of 100. This leads to a recall of 27.2%. The second-best precision of 58.2% can be achieved by removing files which are not connected by *SCS*. This leads to a recall of 45.6%.

Table 9.6. Combination of Improvements

Code Con.	*Freq.*	*Code Struct*	GS Links	Link Cand.		Correct		Wrong		Not Found	Precision		Recall		$F_{0.5}$	F_1
				All	Edit	All	Edit	All	Edit		All	Edit	All	Edit		
True	*20*	*0*	217	124	123	82	82	42	41	135	0.661	0.667	0.378	0.378	0.578	0.482
True	*20*	*4*	217	151	148	101	101	50	47	116	0.669	0.682	0.465	0.465	0.624	0.553
True	*100*	*0*	217	71	71	47	47	24	24	170	0.662	0.662	0.217	0.217	0.469	0.326
True	*100*	*4*	217	87	87	58	58	29	29	159	0.667	0.667	0.267	0.267	0.513	0.382

After the evaluation of the single improvement techniques, the combination of the two techniques with the best results was also investigated. First, the code files which were not connected by *SCS* within a story were removed, and then the remaining interaction links were filter wrt. frequency. Table 9.6 shows the resultant precision of 66.7% for frequency 20 ($F_{0.5}$ is 0.578) and 66.2% for frequency 100 ($F_{0.5}$ is 0.469). So for frequency 100, the precision decreased when one looked at *SCS* connected files. For frequency 20, the best $F_{0.5}$-measure of all evaluations is achieved. As a result of this, *IL's SCS*-based recall improvement (cf. Section 5.2.3.2) was applied to both settings. Again frequency 20 yielded the best results.

Altogether *RQ3* can be answered as follows: with the improvement techniques for detecting wrong links, the precision was improved from 43.0% to 68.2% (increase of 25.2%). On the other hand, the recall decreased from 73.7% without improvement techniques to 46.5%. This yields the best $F_{0.5}$-measure of 0.624.

9.2.3 Discussion

In the following all hypotheses wrt. *IL* and the rationale for the improvement techniques are discussed. The bad level of precision compared to the previous study (cf. Chapter 8) for *IL* clearly indicates that the developers did not use the recording in a disciplined way. The detailed evaluations for the developers did not show big differences, so this finding was true for all three developers.

Several improvement techniques to detect wrong links were tried: a focus on edit interactions, duration, source code owner, source code type and removing of files with many links did not yield considerable precision improvement. Only frequency and removal of links to files which were not connected by *SCS* within a story improved the precision considerably by up to almost 70%, with recall above 45%. For the purposes of direct usage, the resulting links of this evaluation study are not sufficient, as the best achieved precision for IL_i still means that thirty percent of the created links would be incorrect and would hence mislead developers during link usage.

Thus, three further directions of research can be identified. (a) Come up with further improvement techniques to detect wrong links, which yield a precision close to 100%. (b) Try to support the developers in applying interaction recording in a more disciplined way. The results of the previous study on the Mylyn project (cf. Chapter 8) showed that it is possible for developers to use interaction recording in a disciplined way. It could be that students are particularly bad with this discipline. (c) Instead of automatic link creation support, *ILog* could generate links as recommendations to the developers.

In the approach of Delater and Paech [2013], more coarse-grained *VCS* change logs were used to create links and the developers were given different means to create links based on the logs during a sprint or at the end of a project. In relation to this, the *ILog* approach could be used to give recommendations to the developers at different points in the sprint or project for links to create, based on their interactions. Then, developers would have to detect the wrong links themselves. However, this would countervail avoiding any additional overhead by trace link creation for the developers as much as possible.

9.3 Conclusion

In this study the precision and recall of the *ILog's IL* technique for trace link creation in the S_{2017} student project were investigated. Contrary to the previous study (cf. Chapter 8 and Section 8.3), the original *IL* technique only achieved a precision of about 50%. Therefore, several improvement techniques for the detection of wrong links were implemented: a focus on edit interactions, duration, source code owner, source code type and removing of files with many links did not yield considerable precision improvement. Only frequency and removal of links to files which were not connected by *SCS* within a story improved the precision considerably by up to almost 70% with above 45% recall. As discussed in the previous Section 9.2.3, this is not sufficient for the purposes of direct link usage and for the research goal **G1** of the thesis.

Chapter 10

Using Commits and Interaction Logs for Trace Link Creation

In this chapter the third evaluation study of the *ILog* approach is presented. Due to the insufficient results of *ILog's* manual interaction assignment technique *IL* for continuous link creation and direct link usage, even when applying improvement techniques, in the previous study (cf. Chapter 9) the commit-based interaction log assignment IL_{Com} was developed (cf. Section 5.2.1.2). IL_{Com} uses issue IDs which are provided in commit messages to assign interactions to requirements and removes the manual additional effort for developers, as was required in the previous study. All code files touched in the interactions before a commit are associated with the requirement which is identified through the issue ID. The common convention in software development project to provide issue IDs in commit messages (cf. Section 2.2.3) motivated the development of IL_{Com}. Thus, this study answers the general research question: *To what extent can the combination of commit and interaction data improve ILog?*

This chapter is divided into two parts. The first part in Section 10.1 provides a report of a retrospective study in which the application of IL_{Com} was simulated with the S_{2017} dataset from the previous project.

The second part is structured similarly to the previous study chapters. Section 10.2 introduces the experimental design of the study and consists of the detailed research questions, an overview of the experimental activities performed, how trace links have been created with the different trace link creation techniques using the previously introduced S_{2018} dataset, and how the resulting trace links from the different creation techniques have been evaluated. Section 10.3 presents the results by answering the previously stated research questions together with a discussion. Section 10.4 summarizes the results of the study which showed that IL_{Com} achieves very good precision and recall and that it is much better in both precision and recall than other trace link creation techniques.

10.1 Retrospective Study

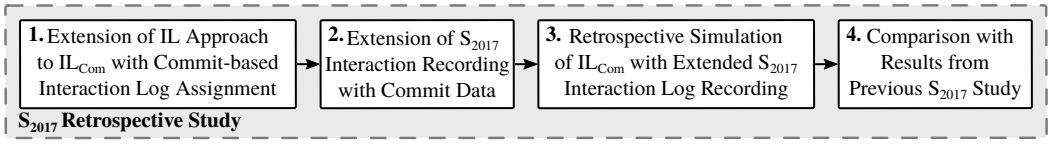

Figure 10.1. Retrospective Study Design: Overview of Activities Performed

Figure 10.1 shows the four activities of the retrospective study in which IL_{Com}, i.e. *ILogs's* commit-based interaction assignment technique, was developed and initially evaluated. The initial motivation to use the developer's commits for interaction assignment came from the common practice to specify issue IDs in commit messages [Rath et al., 2017].

The S_{2017} project dataset was used to simulate the application of IL_{Com} retrospectively. This was possible since the student developers of the S_{2017} project also used issue IDs in their commit messages.

The retrospective simulated application of IL_{Com} with the S_{2017} project data set directly improved the precision from 43.0% to 56.6% without affecting the recall. The further details of the retrospective study are described in the following.

First the S_{2017} project data was analyzed regarding issue IDs in commit messages. It turned out that there were a significantly greater number of commits with issue IDs (per developer) than there were activation and deactivation events for requirements in the recorded interaction logs. For one developer, the processing of 18 requirements was recorded in the interaction logs, but there were 71 commits with requirement issue IDs for the same developer in the Git *VCS* repository. This does not directly indicate that the interaction log recording is wrong, since it is possible that a developer performed multiple commits for one requirement successively. However, after a random check of the time span of interaction recording for two requirements, it turned out that there were commits with different issue IDs in this time span. This encouraged the further data analysis and the retrospective simulation of the application of IL_{Com}.

Table 10.1. S_{2017} Project Retrospective Study: Precision and Recall

Technique	Precision	Recall	$F_{0.5}$	$F_{1.0}$	#Links					#Stories	#Subtasks	Src Files	
					CE	*TP*	*FP*	*GS*	*FN*			*Used*	*GS*
IL	0.430	0.737	0.469	0.543	372	160	212	217	57	19	98	89	91
IL_i	**0.669**	0.465	**0.615**	0.549	151	101	50	217	116	13	72	63	91
ComL	0.620	0.465	0.581	0.532	163	101	62	217	116	19	98	78	91
$ComL_i$	**0.659**	0.401	**0.584**	0.499	132	87	45	217	130	11	66	59	91
IL_{Com}	0.566	0.733	0.593	0.639	281	159	122	217	58	19	98	86	91
IL_{Com_i}	**0.736**	0.539	**0.686**	0.622	159	117	42	217	100	13	72	63	91

Table 10.1 presents the results of the retrospective study together with the data from the S_{2017} dataset. Trace links were created by the different techniques as de-

scribed in the following. For *ComL*, links for all commits with requirement issue IDs in the commit message, from the requirement referenced by the ID to all source code files of the commit, were created. To simulate the application of IL_{Com} retrospectively, the interactions recorded for *IL* and the commits with issue IDs were used. The Git *VCS* commits with requirement issue IDs and the interaction log recording were ordered by time. All interaction log recordings between two commits with issue IDs were assigned to the issue from the second commit. Since there were also commits without issue ID, which were ignored in this evaluation, this type of interaction log recordings for commit assignment is not perfect. If a developer simply forgot to add an issue ID in a commit, the interactions are assigned incorrectly and precision is impaired.

The subscript *i* in the *Technique* column of Table 10.1 indicates the usage of improvement techniques (cf. Section 5.2.3). For *IL* and IL_{Com}, wrong link detection techniques based on interaction data and on *SCS* were used. For *ComL*, only *SCS*-based wrong link detection techniques were used, since there is no interaction data in this link creation technique (cf. Section 7.5). Table 10.1 always shows the best achieved $f_{0.5}$-measure within all performed settings for any single given technique. Moreover, the overall best values for precision and $f_{0.5}$-measure are highlighted. IL_{Com_i} has a precision of 73.6%, a recall of 53.9% and a $f0.5$-measure of 0.686, which outperforms the precision and recall of all other approaches. This confirmed the idea that *IL* can be combined with the use of issue IDs from commit messages to assign recorded interactions to requirements.

10.2 Experiment Design

In this section the details of the studies experimental design, as shown in Figure 10.2, are described, in particular wrt. usage of the S_{2018} student project dataset and the application of IL_{Com} (cf. Section 5.2.1.2). Due to the positive results of IL_{Com} in the retrospective study, a new study in which IL_{Com} with commit-based interaction assignment was directly applied by the student developers was set up within the S_{2018} project.

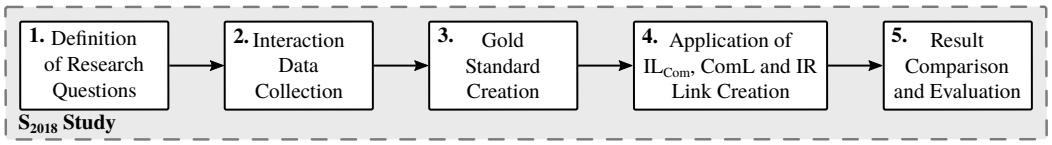

Figure 10.2. 3. Study Experimental Design: Overview of Activities Performed

10.2.1 Research Questions

The overall research question *To what extent can the combination of commit and interaction data improve ILog?*, which is answered in this study, is divided into the following three sub-questions:

RQ1: *What is the precision and recall of IL_{Com}- and IL_{Com_i}-created trace links?* The hypothesis was that the initial precision of IL_{Com} improves, compared to the previous S_{2017} study (cf. Section 9.3), since there is no additional effort for requirement selection by developers. For IL_{Com_i}, when compared to IL_{Com}, a further precision improvement was expected.

RQ2: *What is the precision and recall of ComL- and $ComL_i$-created trace links?* The hypothesis was that precision and recall are worse than the precision of IL_{Com}- and IL_{Com_i}-created links, respectively, as the latter uses more information, in the form of the interactions.

RQ3: *What is the precision and recall of IR- and IR_i-created trace links?* The hypothesis was that *IR* has a significantly worse precision and a similar recall in comparison to IL_{Com}.

The overall goal of this study is to evaluate whether the interaction and commit-based link creation by IL_{Com} improves the precision compared to the only interaction-based link creation by *IL* (*RQ1*). Moreover, it is investigated whether recording and using interactions outperforms link creation, which relies on commit data only (*RQ2*). Finally, the results of IL_{Com}-created links are also compared with *IR*, since *IR* serves as a baseline for automated link creation and for the purposes of comparison with the previous studies (*RQ3*).

10.2.2 Trace Link Creation

Before the creation of trace links with the different trace link creation techniques, interactions were recorded with the commit-based interaction assignment tool (cf. Section 5.2.1.2) during the S_{2018} project. Moreover, before the project was completed, the developers created a gold standard for the trace links (cf. Section 7.3).

After the interaction recording and gold standard creation, trace link were created for the S_{2018} project with IL_{Com}, *ComL* and *IR*, further improvement techniques were applied to all three link creation techniques. For IL_{Com} the interaction recordings were assigned by 205 commits to the 17 stories, and for IL_{Com_i} the final version of the source code files from the Git repository were also used for link creation. For *ComL* the 395 commits of the Git repository and for $ComL_i$ the final version of the source code files from the Git repository were used for link creation. For *IR* link creation the text of the 17 stories, 74 sub-tasks, and the final version of the source code files from the Git repository were used. As in the previous studies

IR link creation was performed with both *IR* techniques *VSM* and *LSI* including preprocessing (cf. Section 2.2.2.1). In addition to the initial *IR*-based link creation, the final version of the source code files from the Git repository was also used for IR_i. Finally, the trace links of all trace link creation techniques were evaluated with the previously created gold standard.

10.3 Results

This section reports the results of the evaluations performed and answers the research questions.

10.3.1 Commit-based Interaction Assignment – IL_{Com}

Table 10.2. Results for IL_{Com} and IL_{Com_i} with Different Settings

Technique	Setting*	Precision	Recall	$F_{0.5}$	$F_{1.0}$	#Links					Src Files	
						CE	*TP*	*FP*	*GS*	*FN*	*Used*	*GS*
IL_{Com}	none	0.849	0.673	0.807	0.751	245	208	37	309	101	58	66
IL_{Com_i}	*T:e*	0.904	0.460	0.758	0.609	157	142	15	309	167	58	66
IL_{Com_i}	*T:s*	0.829	0.282	0.597	0.420	105	87	18	309	222	37	66
IL_{Com_i}	*D10*	0.885	0.521	0.776	0.656	182	161	21	309	148	52	66
IL_{Com_i}	*D60*	0.901	0.411	0.727	0.564	141	127	14	309	182	50	66
IL_{Com_i}	*F2*	0.813	0.463	0.706	0.590	176	143	33	309	166	54	66
IL_{Com_i}	*F10*	0.850	0.311	0.631	0.455	113	96	17	309	213	40	66
IL_{Com_i}	*Sis*	0.904	0.485	0.771	0.632	166	150	16	309	159	40	66
IL_{Com_i}	*T:e,s; Sis;CS*	**0.900**	0.790	**0.876**	0.841	271	244	27	309	65	62	66

* *Notation for the used improvement techniques settings: T:e|s = Type:edit|select, D10|D60 = duration >= 10|60 sec., F2|10 = frequency >= 2|10, Sis =* SCS *in story, CS =* SCS *recall improvement*

Table 10.2 presents the results for IL_{Com} and for the different improvement technique settings used for IL_{Com_i} (cf. Section 5.2.3). IL_{Com} has a precision of 84.9% and a recall of 67.3% and thus a $f_{0.5}$-measure of 0.807. Similarly to the previous study with the S_{2017} dataset (cf. Section 9.2.2), different settings for the improvement techniques were evaluated, as denoted in the *Setting* column of Table 10.2. Initially, the different improvement techniques were investigated in isolation, and then different techniques were combined to achieve the overall best precision improvement. On this best precision result, *SCS*-based recall improvement was applied as well. The last row of Table 10.2 shows this best case of IL_{Com_i}. For this case, the setting was to use the interaction types select and edit (T:e,s), to restrict the source code files to be connected with each other by *SCS* in the story (Sis) and to use the *SCS* to improve recall (CS). In this best case, IL_{Com_i} has a precision of 90.0% and a recall of 79.0%, and thus a $f_{0.5}$-measure of 0.876. Thus, IL_{Com_i} improves precision by 5.1%, recall by 22.7% and $f_{0.5}$-measure by 0.069 compared to IL_{Com}.

10.3.2 Comparison of IL_{Com} and ComL

TABLE 10.3. RESULTS FOR *ComL*, $ComL_i$ AND COMPARISON WITH IL_{Com} AND IL_{Com_i}

Technique*	Precision	Recall	$F_{0.5}$	$F_{1.0}$	#Links					Src Files	
					CE	*TP*	*FP*	*GS*	*FN*	*Used*	*GS*
IL_{Com}	0.849	0.673	0.807	0.751	245	208	37	309	101	58	66
IL_{Com_i}	**0.900**	0.790	**0.876**	0.841	271	244	27	309	65	62	66
ComL	0.668	0.417	0.597	0.514	193	129	64	309	180	59	66
$ComL_i$	**0.675**	0.443	**0.611**	0.535	203	137	66	309	172	61	66

* For the application of improvement techniques the best case is shown

Table 10.3 presents the results for *ComL* and $ComL_i$. For the sake of comparison, the previously reported results of IL_{Com} and IL_{Com_i} are also presented. *ComL* has a precision of 66.8% and a recall of 41.7% and thus a $f_{0.5}$-measure of 0.597. For $ComL_i$, the *SCS in story* precision improvement was first applied followed by *SCS* recall improvement. $ComL_i$ has a precision of 67.5% and a recall of 44.3% and thus a $f_{0.5}$-measure of 0.611. In comparison to IL_{Com} and IL_{Com_i}, precision, recall, and $f_{0.5}$-measure are all worse, respectively.

10.3.3 Comparison of IL_{Com} and IR

TABLE 10.4. RESULTS FOR *IR*, IR_i AND COMPARISON WITH IL_{Com} AND IL_{Com_i}

Technique*	Precision	Recall	$F_{0.5}$	$F_{1.0}$	#Links					#Stories	Src Files	
					CE	*TP*	*FP*	*GS*	*FN*		*Used*	*GS*
IL_{Com}	0.849	0.673	0.807	0.751	245	208	37	309	101	17	58	66
IL_{Com_i}	**0.900**	0.790	**0.876**	0.841	271	244	27	309	65	17	62	66
IR	0.335	0.492	0.358	0.398	454	152	302	309	157	16	60	66
IR_i	**0.369**	0.557	**0.396**	0.444	466	172	294	309	137	16	64	66

* Used *IR* settings denoted as *<IR-technique(similarity threshold)>: VSM (0.2)*

Table 10.4 prsents the results for *IR* and IR_i. For the sake of comparison the previously reported results of IL_{Com} and IL_{Com_i} are also presented. As *IR* technique *VSM* with a similarity threshold of 0.2 was used. *IR* has a precision of 33.5% and a recall of 49.2% and thus a $f_{0.5}$-measure of 0.358. For IR_i, the *SCS in story* precision improvement was first applied followed by *SCS* recall improvement. IR_i improves precision by 3.4%, recall by 6.5% and $f_{0.5}$-measure by 0.038, when compared to *IR*. In comparison to IL_{Com} and IL_{Com_i}, precision, recall, and $f_{0.5}$-measure are all worse, respectively.

10.3.4 Discussion

Precision and recall of IL_{Com} are better than *IL* for both the S_{2017} and S_{2018} datasets. When looking at all three evaluation studies, it can be seen that *ILog's IL* and IL_{Com} technique outperform all other link creation techniques, i.e. *IR*- and commit-based link creation *ComL* (cf. Section 11.2). The fact that *IR* link creation between unstructured requirements in *ITS* and source code is worse than in structured requirement cases is a finding which has also been reported by others [Hayes et al., 2006, Borg et al., 2014, Merten et al., 2016b]. This is also confirmed by the three studies and was one of the initial motivations for the development of *ILog*.

There are several possible reasons for the worse behaviour of *ComL* in comparison to IL_{Com}. It is interesting that the precision of *ComL* is roughly 60% in the retrospective study with the S_{2017} dataset and in this study with the S_{2018} dataset. This result means that the issue IDs given by the developers are only partly correct. This observation is similar to research within developers' commits behavior and the contents of commits [Herzig and Zeller, 2013, Kirinuki et al., 2014]. These studies have reported on tangled changes, in which a commit often comprises multiple unrelated issues. Also, it was observed that developers manually excluded files in one commit, which were correct in the gold standard, and then included these files in a follow-up commit. One reason for this behavior could be a change of the requirement during the project time. Thus, the exclusion behavior was correct when the commit was performed, but was incorrect for the final state of the requirement. The reason for the worse recall of *ComL* in comparison to IL_{Com} could be select interactions. Select interactions are not detected by commits. The files missed based on select interactions also affect the application of *SCS*-based recall improvement.

The *ILog* improvement techniques initially developed in the previous studies also proved to be reasonable in this study. Moreover, the improvement techniques also performed well for links created with *IR* and *ComL*. By applying the improvement techniques to detect wrong links, the precision is improved, independent of how the links were created. As improvement techniques impair recall, *SCS*-based recall improvement was applied. The improvement of recall by using the *SCS* worked reasonably well for *ILog's IL* technique in the last two studies, and it is outperformed by IL_{Com} in this study. The application of recall improvement in this study resulted in the best overall recall for the complete studies.

Altogether, this study shows that the creation of links with interaction and commit data by IL_{Com_i} achieves very good precision and recall. This confirms the assumption that the additional effort of manually selecting the requirement to work on caused the bad precision of *IL* in the previous S_{2017} study (cf. Section 9.3). It is very likely that precision and recall can be even better, if developers directly use the created links during the projects, as in the Mylyn project. The use will likely motivate developers to use interaction logging and commit IDs carefully.

10.4 Conclusion

This study investigated the precision and recall of *ILog's* IL_{Com} technique. In contrast to the previous studies, instead of manual interaction log assignment a commit-based interaction log assignment was used. Through the usage of commit-based interaction log assignment, the additional effort for developers to assign interaction log recordings to requirements was reduced and the need for interaction log recording awareness was removed.

IL_{Com} builds on the common practice of specifying issue IDs in commit messages. It uses these issue IDs from commit messages to assign interaction log recording to requirements. IL_{Com} has a precision of 90.0% and recall of 79.0%, which outperforms the results of the previous S_{2017} study (precision of 68.2% and recall of 46.5%, cf. Section 9.3). Thus, precision is not perfect, but it is likely that this is a very good basis for continuous link creation and usage, as defined in research goal **G1**. Furthermore, IL_{Com} is also applicable where developers are not particularly interested in interaction recording. The study showed that IL_{Com} outperforms *IR* and purely commit-based linking and is superior to current machine learning-based approaches as well [Rath et al., 2018]. Clearly, it is interesting to confirm this with further studies and to study whether this also holds for more structured requirements in which *IR* is typically used.

Chapter 11

Discussion

This chapter discusses the overall results of all performed *ILog* evaluation studies. In Section 11.1 the study's threats to validity are discussed. In Section 11.2 the overall results of all three studies are summarized and discussed in connection with the thesis' research goals as a whole.

11.1 Threats to Validity

The limits of empirical research regarding its internal and external validity are a well known and considered to be of an acceptable constraint. Although known threats were considered in the evaluation studies of this thesis, general and perfect validity is not possible. Thus, this Section introduces the threats to the validity of empirical studies as were proposed by Runeson and Höst [2008], Wohlin et al. [2012] and Yin [2018], followed by a discussion of the threats to validity of the *ILog* evaluation studies. Runeson and Höst suggest a scheme for discussing treats to validity which distinguishes between four aspects of the validity, and which is summarized in the following:

Construct Validity discusses whether the measures used, e.g. precision, recall etc., reflect what is being investigated in the research questions of the study.

Internal Validity discusses whether different assumptions have been made for the participants of the studies, e.g. for rating link candidates during gold standard creation.

External Validity discusses to what extent the results are generalizable, e.g. whether, if the experiment of an *ILog* evaluation study is replicated by others in a different context, the results should be similar.

Reliability discusses whether the study can be replicated in a reliable way, e.g. whether, when using the same set-up and data, the results should be the same if performed by others.

The internal validity of the first *ILog* evaluation study with the Mylyn project (cf. Chapter 8) is threatened since the manual validation of trace links was only performed by the author of this thesis. However, the author is very familiar with the Mylyn project in general, its source code, the development infrastructure used, and has of over ten years of Mylyn-specific development experience. The internal validity of the two evaluation studies with the S_{2017} (cf. Chapter 9) and the S_{2018} (cf. Chapter 10) student projects is threatened since the manual validation of trace links in the gold standard was performed by the students working as developers in a project context of the research group of the thesis author. However, this ensured that the experts created the gold standard. Also, the evaluation of the links was performed at the end of the projects so that there was no conflict of interest for the students to influence their grading.

When comparing the results achieved with the *ILog* approach to *IR*, the set-up of the *IR* techniques is a crucial factor. With respect to preprocessing, all common steps have been performed including identifier splitting, which is specific to the datasets used. However, the low threshold values impair the results for the precision of *IR*. Therefore, further comparison of *ILog* and *IR*, in which higher threshold values are possible (e.g. in a context with more structured requirements descriptions), is necessary. Applying *SCS*-based recall improvement on *IR* created links which already have had bad precision further impairs the results of *IR*. However, applying *SCS*-based precision improvement before *SCS*-based recall improvement also resulted in better precision for *IR*-created links and showed the general applicability of *SCS*-based improvement for trace links.

The external validity depends on the availability of interaction logs and the respective tooling and usage of the tooling by developers. By *ILog's* commit-based interaction log assignment IL_{Com}, there is no additional burden for developers despite the initial set-up and ideal motivations. The generalizability of the results based on three projects is limited. However, *ILog* worked well in two different project types: firstly in the loosely organized and structured open source project Mylyn, and secondly in two student projects following the Scrum project paradigm. Thus, it can be expected that *ILog* will achieve even better results when applied in a more strictly organized and structured industry project.

In the Mylyn open source project, the developers used their own implemented interaction logging approach and thus worked in this in a very disciplined way. It is very likely that the student developers of the S_{2017} project with manual interaction log assignment did not apply the interaction logging in an as disciplined way as the Mylyn developers, since they had no awareness of it. This assumption is verified to a certain extent by the S_{2018} project. Although explicitly requested, not all commits in the S_{2018} project contained a Jira issue ID in the commit messages. This affects the resulting assignment of recorded interaction logs to requirement issues and thus the created trace links. However, the percentage of commits with issue IDs is similar

to that which has been reported for other projects [Rath et al., 2017]. This indicates that the results of *ILog's* evaluation might also apply for industry projects.

11.2 Evaluation Studies Summary

Table 11.1. Results for IL, IL_{Com}, $ComL$ and IR in all Studies

Technique[1]	Data Set	Precision	Recall	$F_{0.5}$	$F_{1.0}$	#Links[2]					#Stories	Src Files	
						CE	*TP*	*FP*	*GS*	*FN*		*Used*	*GS*
IL_i	M_{2007}	1.000	0.929	0.985	0.963	2565	2565	0	2761	196	50	627	627
	M_{2012}	1.000	0.800	0.952	0.889	1126	1126	0	1408	282	50	363	702
	S_{2017}	0.682	0.465	0.624	0.553	148	101	47	217	116	13	63	91
IL_{Com_i}	S_{2017}	0.736	0.539	0.686	0.622	159	117	42	217	100	13	63	91
	S_{2018}	0.900	0.790	0.876	0.841	271	244	27	309	65	17	62	66
$ComL_i$	S_{2017}	0.659	0.401	0.584	0.499	132	87	45	217	130	11	59	91
	S_{2018}	0.675	0.443	0.611	0.535	203	137	66	309	172	17	61	66
IR_i	M_{2007}	0.386	0.440	0.396	0.411	3143	1214	1929	2761	1547	41	308	627
	M_{2012}	0.283	0.557	0.314	0.376	2766	784	1982	1408	624	35	354	702
	S_{2017}	0.351	0.217	0.312	0.268	134	47	87	217	170	9	21	91
	S_{2018}	0.369	0.557	0.396	0.444	466	172	294	309	137	16	64	66

[1] *IR* settings for the data sets are denoted as *<IR-technique(similarity threshold) >*: M_{2007} *LSI(0.3)*, M_{2012} *LSI(0.5)*, S_{2017} *LSI(0.1)*, S_{2018} *LSI(0.2)*

[2] *created (CE)*, *True Positive* (*TP*) ≙ correct, *False Positive* (*FP*) ≙ wrong, *Gold Standard* (*GS*), *False Negative* (*FN*) ≙ not found

In three evaluation studies (cf. Chapters 8, 9 and 10), different techniques of the *ILog* approach were evaluated. Table 11.1 presents an overview of the achieved study results of the *ILog* techniques *IL* and IL_{Com} and the other evaluated trace link creation techniques *ComL* and *IR*-based trace link creation. The numbers shown are always the best achieved results, which include the application of improvement techniques (cf. Section 5.2.3).

The datasets M_{2007} and M_{2012} were used in the first evaluation study (cf. Chapter 8), in which the interaction data recorded in the Mylyn Open Source project was used and the general practicality of *ILog* was evaluated.

The dataset S_{2017} was used in the second evaluation study (cf. Chapter 9). This dataset was created in an student project using the manual interaction assignment technique *IL* of *ILog*. Due to the initially unsatisfying precision of *IL*-created links in the second evaluation study, improvement techniques to detect wrong links were developed and evaluated as well. The interaction data-based improvement techniques worked well together with *IL* and IL_{Com} with the S_{2017} and S_{2018} datasets, inasmuch as precision was noticeably improved with limited effects on recall. *SCS*-based improvement techniques worked well for all trace link creation techniques in all the studies performed and the datasets used. Thus the studies also showed the general applicability of the developed improvement techniques.

The dataset S_{2018} was used in the third evaluation study (cf. Chapter 10). The S_{2018} dataset was created in another student project, using the commit-based

interaction assignment technique IL_{Com}. Before the actual application of IL_{Com} in the S_{2018} student project, the S_{2017} dataset from the previous second study was used to simulate the application of IL_{Com}. The results achieved in the simulated application of IL_{Com} were better than the original results achieved with IL. As can be seen for the IL and IL_{Com} S_{2017} results, which are shown in Table 11.1, IL achieved a precision of 68.2%, a recall of 46.5% and a $f_{0.5}$-measure of 0.624 in the shown best case, whereas the simulation of IL_{Com} achieved a precision of 73.6%, a recall of 53.9% and a $f_{0.5}$-measure of 0.686. The achieved improvement of simulated IL_{Com} application in comparison to IL also motivated the third evaluation study.

When comparing the precision, recall and $f_{0.5}$-measures of IL and IL_{Com} with the other link creation techniques IR and $ComL$, the two $ILog$ techniques clearly outperform the other link creation techniques in all studies and datasets. When considering the fulfilment of the thesis' research goal **G1**, which was to continuously create trace links for direct usage, the precision values of *ILog's* IL technique achieved with the M_{2007} and the M_{2012} datasets and the values of *ILog's* IL_{Com} technique achieved with the S_{2018} dataset are the only values which are acceptable for this goal. Even in the best case of other approaches, such as $ComL$ in the S_{2018} project with its precision of 67.5%, still about one third of the created links are wrong.

The unsatisfying results for IL in the second study with the S_{2017} dataset could be balanced to some extent by the developed improvement techniques, which use the interaction data-specific attributes of the created trace links and SCS. However, even with applied improvement techniques the achieved IL precision for the S_{2017} dataset of 68.2% is still not practical for directly using the links. However, by replicating the context of the second study, i.e. a Scrum project with student developers, in the third study, and by using IL_{Com} with commit-based instead of IL with manual interaction assignment, the results were much better. Thus, comparing the results for IL and IL_{Com} with the S_{2017} and S_{2018} datasets also confirmed that manual interaction assignment had a strong negative impact on the results for IL with the S_{2017} dataset.

For IR-based link creation the two most common IR techniques, VSM and LSI, were used [De Lucia et al., 2007, Borg et al., 2014, Cleland-Huang et al., 2014]. Moreover, common and dataset-specific preprocessing techniques were applied and different similarity threshold values were used to achieve the best possible results. As it can be seen in Table 11.1, the IR results are quite similar in the different datasets and they are also similar to other studies in the context of unstructured requirements in ITS [Merten et al., 2016b]. However, even with the best precision value of 38.6% in the M_{2007} dataset, almost two thirds of all created links are wrong. Thus, as expected, IR is not capable of continuously creating and providing links for direct use, as it was targeted by research goal **G1**. Clearly, it is interesting to confirm this with further studies and to study whether this also holds for more structured requirements in cases where IR is typically used.

Part V

Conclusion

Chapter 12

Summary

This thesis contributes to the body of knowledge in *SE* and *RE* with respect to automatic trace link creation and maintenance. For the research goal **G1**, which was to continuously create trace links with perfect precision and to make the created links directly usable, the review of existing trace link creation approaches which was presented in Chapter 3 showed that all reviewed approaches are not fully capable for this purpose. Often the usage scenario of approaches is only vaguely defined, and if the usage of links is discussed, the authors most often refer to the standard scenarios of trace link usage, such as verification purposes and the therefore manual triggered link creation. More important for the research goal **G1** is the insufficient precision and recall of created links in the existing approaches. Most of the approaches are optimized towards recall, which is to say that they try to create as many correct links as possible, and they sacrifice the precision of the resulting links. As a result, the further assessment of resulting link candidates is necessary before their actual usage in the approaches.

For the research goal **G2**, which was to maintain created links along with changes in linked artefacts in Chapter 4, a *SLR* of existing *TM* approaches showed that the maintenance of existing links along with artefact changes is often performed semi-automatically. Only a view of the reviewed *TM* approaches are completely automated. The approaches found were described with a standardized *TM* process, which consists of an impacted link detection and an execution of link change part. In the *TM* approaches reviewed, automation is implemented by rules which often utilize the artefact type but which also require manual user interactions in order to perform the necessary link changes.

Based on these findings and in order to comply the research goals **G1** and **G2**, the *Interaction Log Recording-based Trace Link Creation* (*ILog*) approach was designed and evaluated as the main contribution of this thesis. Chapter 5 introduced the details of the *ILog* approach and Chapter 6 discussed the extension of *ILog* with *TM* capabilities. In three evaluation studies (cf. Chapters 8, 9 and 10), the different

techniques and different aspects of the *ILog* approach were evaluated. The results of the studies showed that *ILog* is capable of creating trace links with very good precision and good recall in different project contexts, and that it can be applied continuously during a project without requiring any manual effort from developers.

ILog uses developers' interactions recorded in an *IDE* with source code files while they work on a requirement for trace link creation. *ILog* consists of three general steps: (1) interaction recording and assignment, (2) trace link creation, and (3) trace link improvement. In the finally evaluated IL_{Com} technique of *ILog*, interactions are assigned to requirements using issue IDs which are provided in *VCS* commit messages. Thus no additional effort is required for trace link creation. Trace link creation is performed by aggregating the recorded interactions using interaction data. For trace link improvement, first precision is improved using the interaction data of trace links and *SCS*, and second recall is also improved using *SCS*.

ILog's usage of developers interactions instead of artefact contents counteracts the quality problems regarding precision and recall of existing *IR*-based and other trace link creation approaches. This is achieved by decoupling *ILog's* link creation from the artefacts' contents and by using the expert knowledge of developers, which is covered by their interactions, instead.

To support the maintenance of links along with changes in linked artefacts in *ILog*, the results of the *TM SLR* of Chapter 4 were used. The standardized *TM* process was utilized to describe *TM* in *ILog*, by integrating *TM* capabilities from two of the reviewed approaches. Since the *TM* capabilities integrated are fully automated and all necessary data sources required are provided by *ILog*, this facilitates a seamless integrated and fully automated *TM*, along with changing artefacts in *ILog*.

Altogether, this thesis shows that the *ILog* approach developed is capable of continuous trace link creation for directly usable trace links. The IL_{Com} technique is also applicable in situations where developers are not particularly interested in interaction recording. In all the studies performed, all other trace link creation techniques were outperformed by *ILog*. Furthermore, by integrating *TM* capabilities which were found in the performed *TM SLR*, and by using the developed generic *TM* process, the general applicability of *ILog* for *TM* along with the changing artefacts is shown.

Chapter 13

Future Work

Based on the evaluation results and state of the *ILog* approach as presented in this thesis, there are several directions for future work. These directions can be separated into work which is focussed on continuing the development of the *ILog* approach and more general research directions in the context of traceability and the usage of interactions.

In order to continue the development of the *ILog* approach, there are obvious next steps arising from the evaluation study results and the *ILog* implementation. In summary these next steps comprise the implementation and evaluation of link maintenance in *ILog*, as discussed in Chapter 6, the implementation and evaluation of link usage, e.g. as discussed in Section 10.4 of the third *ILog* evaluation study, and an evaluation and comparison of *IR* and *ILog* in a project context with more structured requirements, e.g. as discussed in Section 8.3 of the first and Section 10.4 of the third *ILog* evaluation study. In the following these future directions will be illustrated in more detail.

Implementation and Evaluation of Maintenance

As outlined in Chapter 6, the *TM* capabilities of the approaches P08 of Ghabi and Egyed [2012] and P13 of Rahimi et al. [2016] are well suited for the integration in *ILog*. P08 uses *SCS* and P13 refactoring specific rules in order to perform *TM*. The challenge for the implementation of these rules in *ILog* is that there is no implementation available, and thus for the practicability of the implementation, one's own implementation in *ILog* is necessary.

For the integration of P08 in *ILog*, two scenarios are reasonable. In the first integration scenario, the *TM* rules of P08 would replace *ILogs's SCS*-based improvement techniques (cf. Section 5.2.3). The effect of this scenario can be evaluated with all datasets from the *ILog* evaluation studies. For the second integration scenario, the P08 approach would be used to supplement *ILog's SCS*-based improvement techniques. Also for this second integration scenario, an evaluation with existing *ILog*

study data is possible. Since the algorithm to determine whether *TM* is necessary is based on a set of rules in the P08 approach, it would also be possible to perform a more fine-grained evaluation, i.e. to use only a sub-set of the rules in *ILog*.

Also for the integration of P13 in *ILog*, two options are reasonable. In the first option, the approach would be applied as intended by using two versions of source code files or requirements in order to detect a certain refactoring and then to execute the respective *TM* rules. In the second option, the refactorings could be detected by using interaction data either by directly detecting and evaluating refactoring events or by defining sequences of basic events which correspond to certain refactorings. An evaluation of the effects when integrating the P13 approach in *ILog* is also possible with the existing evaluation study data. For this, it would be necessary to create a second-to-last version of links with *ILog*, e.g. by using only the second-to-last commit with an issue ID for link creation. As a next step, it would be possible to maintain the *ILog* links created for the second-to-last commit and to compare the resulting maintained links with the gold standard and with *ILog* links created for the last commit. Clearly the maintenance evaluation options which are here described are only a simulation of link maintenance with *ILog*. However, the evaluation of the different options would help to discover a suitable setting for *TM* in *ILog*. This setting could then be used in a new study in which *TM* is performed during the study.

Implementation and Evaluation of Link Usage

In all three *ILog* studies the links were only created after the projects were completed and used for the evaluation of *ILog*. After improving the *ILog* approach with improvement techniques and commit-based interaction assignment throughout the studies which were already performed, to the final IL_{Com} version of *ILog*, a next step would be to provide the created links to the developers during a further evaluation project.

There are two reasonable ways for providing *ILog*-created links during a project to the developers. The first is within the project's *ITS* and the second is within the *IDE*. In order to provide *ILog*-created links in an *ITS*, a Jira-specific implementation already exists. This Jira plug-in uses interaction data which is attached to issues, previously uploaded by the Eclipse interaction capturing plug-in, in order to perform *ILog*-based link creation, and it provides the resulting links in an additional panel shown in the issue's detailed view. For both *IDEs* (Eclipse and IntelliJ) used in the evaluation projects, the implementation of plug-ins which provide links to the developers is also possible with manageable effort. For this, the interaction-capturing plug-ins (cf. Section 5.2.1) which were already implemented can be extended with functionality in order to directly show created links to developers. In addition, both *IDEs* have capabilities for creating connections to various *ITS* such as Jira, so that

providing links which are created due to the interactions by one developer to another is relatively simple to implement. The Eclipse *IDE* interaction-capturing plug-in, which was used in the third *ILog* study to evaluate the commit-based interaction assignment technique IL_{Com}, already uses the *ITS* connection capabilities of Eclipse in order to upload recorded interactions to Jira issues (cf. Section 5.2.1.2).

In order to evaluate how directly providing *ILog* links to developers affects the performance of *ILog*, different experimental set-ups are possible. The options to provide the links in the *IDE* or in the *ITS* can be evaluated both in combination and in isolation. Furthermore, it is also possible to group developers who participate in the usage evaluation, between one group who use the links and another group which does not. Within such a link usage study, it would also be possible to evaluate the assumed positive impact on *ILogs's* link quality, since if developers discover the positive effects of link usage, the quality of *ILog* links will increase further.

Evaluation and Comparison with IR in a Structured Requirement Context

The three performed *ILog* evaluation studies were all performed with in projects using unstructured requirements. That is to say, the requirements were specified as text in issues which were managed in an *ITS*. Even though projects based around an *ITS* that use unstructured requirements are the intended usage context for *ILog*, it would also be interesting to evaluate *ILog* in comparison with *IR* in a more structure requirements context. Typically, *IR* is used for trace link creation in more structured project contexts as in the automotive or the aeronautic industry, where the requirements are managed with a distinct requirements tool. However, these more structured projects also come with strict guidelines how developers have to perform implementation tasks for the specified requirements. Thus, it can also be assumed that the quality of *ILog* links will not decline in such a structured context.

Traceability and Interaction Data Usage

Apart from the future directions which have just been presented for the continuation of *ILog* development and research, there are also more general possible research directions for traceability and interaction research. These will be outlined in the following paragraphs.

In general, *ILog* opens up new possibilities for automatic trace link creation during software development projects and challenges existing artefact content-based approaches. Even though the problems of using an artefact's textual contents for trace link creation are well known, traceability research relies on these textual content-based techniques as the core of automatic link creation. Non-content-based techniques are often only used to improve links which were initially created or which were created along with the textual content of artefacts. Clearly the results of this thesis have shown that stepping away from this paradigm can result in trace links

which possess a very good precision and recall, which is indeed as good as for existing techniques. For further research in the usage of interactions for trace link creation, it is possible to include *ILog*-based trace link creation without much additional effort in trace link-related studies. After an initial set-up no further effort is required to deploy *ILog* in a project. This enables the gathering of further data for *ILog* and the use of the *ILog* results as a measure of comparison.

ILog and its evaluations have also shown the practicability of using interaction data which was created by developers and which is also compliant to, as it has been reported by others (cf. Section 2.1.4.2). As was particularly present in the second study, the concern of dealing with noise, i.e. interactions which are wrong in a given context, is an important aspect when using interaction data. However, in all three studies the noise could be cleared either completely or at least to a reasonable extent, initially by basic filtering, such as by the limitation of interactions to more mature files, in which case files are also contained in a *VCS*. These *ILog* noise detection principles could also be adopted by other domains and tasks which are different from trace link creation but also use interaction data, such as recommendation system, impact analyses, classification, etc. Furthermore, sophisticated interaction noise principles could be researched for the adoption in *ILog* from those domains and tasks as well.

Overall, the results achieved with the usage of interaction data in *ILog* are promising and have enabled for the continuously creation of trace links and for direct use during a project. Furthermore, the *ILog* approach shows that automatic *TM*, along with changing artefacts, can be implemented.

Part VI

Appendix

Appendix A

Supplementary Material for Trace Link Maintenance SLR

This appendix provides additional details for the *Systematic Literature Review* (*SLR*) about *Trace Link Maintenance* (*TM*) described in Chapter 4. Section A.1 provides more details about how the search for publications was performed. Section A.2 provides more detailed textual descriptions of the reviewed *TM* approaches, structured wrt. the standardized *TM* process (cf. Section 4.3).

A.1 Publication Search

This section provides additional details about the publication search. Section A.1.1 presents details about the performed keyword pre-search. Section A.1.2 presents details about the scientific database specific query adoptions, which were necessary due to technical concerns of the search interfaces. Section A.1.3 describes how the resulting publications were filtered to determine a primary publication for the *TM* approaches described in multiple publications.

A.1.1 Keyword Pre-Search

The idea of performing an initial pre-search is on the one hand to identify the potential amount of publications for a research subject and and on the other hand to check and optimize search terms. To create an initial overview of articles *Google Scholar* has been used with the search term based query shown in Listing A.1.

Listing A.1. Keyword Pre-Search Search Query

```
"traceability maintenance" OR "trace link maintenance"
```

The language for the results has been restricted to English. This resulted in a list of 316 publications. This list was filter based on the following exclusion criteria:

- wrong field/ out of scope, this remove articles from other fields like biology
- to old, articles before the year 2000 have been removed
- not peer reviewed, articles like technical reports and thesis have been removed

In consequence the articles have been evaluated as relevant or not based on their title, i.e. the title had to state about traceability and the maintenance of trace links. If a judgement by only reading the title was not possible, the abstract of the article has been considered as well. Finally this resulted in 51 relevant articles.

A.1.2 Scientific Database Specific Query Adaption

In the following the database specific adoptions of the search query due to technical concerns (i.e. search interface options) is described. This also includes the explanation how the final number of publications found and used publications from each scientific database were achieved.

ACM Digital Library

For the *ACM Digital Library*[1] it is necessary to specify the attributes of articles considered by the key word query. The reason for this is that by default ACM Digital Library search only considers meta-data, like the titles articles, the journal or conference etc., and not the content of the articles. Thus the following Listing A.2 shows the used keyword query for ACM Digital Library.

Listing A.2. ACM Digital Library Key-Word Query

```
((acmdlTitle:(+traceability +maintenance) OR acmdlTitle:(+trace
    +link +maintenance))
OR
(recordAbstract:(+traceability +maintenance) OR recordAbstract
    :(+trace +link +maintenance))
OR
(keywords.author.keyword:(+trace +link +maintenance) OR
    keywords.author.keyword:(+traceability +maintenance)))
AND
(content.ftsec:(+traceability +maintenance) OR content.ftsec:(+
    trace +link +maintenance))
```

The query consists of two basic parts connected by conjunction. The first part searches for the key word combinations of the base query (cf. query in Listing A.1) in the meta-data attributes title (*"acmdlTitel"*), abstract (*"recordAbstract"*) and keywords (*keywords.author.keyword*). In addition it ensures that only results containing at least all key words from one of the parts of the basic query are returned (by the $(+ < term_1 > + < term_n >)$ query syntax) , e.g. only articles which contain

[1] `https://dl.acm.org/advsearch.cfm`

the terms *"traceability"* and *"maintenance"* are returned, articles only containing either one these terms are not returned.

The second part of the query does the same thing as the first but for the content of the article (*"content.ftsec"*) instead of meta-data attributes. Together this results in publications which contain the terms of the basic query in one of the specified meta-data attributes and in the content. This initially resulted in 64 publications received from the ACM Digital Library. After evaluating the articles title and conference or journal series finally 33 results of the ACM Digital Library have been kept.

IEEE Xplore Digital Library

For *IEEE Xplore Digital Library*[2] the command search interface with the setting *Full Text & Metadata* has been used. This was necessary since using the IEEE Xplore Digital Library standard search would restrict the search only to articles meta-data. In addition the used query consists of two basic parts connected by conjunction to ensure that the keyword terms are contained in the meta-data and in the content of the articles. The following Listing A.3 shows the used keyword query for IEEE Xplore Digital Library.

Listing A.3. IEEE Xplore Digital Library Key-Word Query

```
((("Document Title":traceability maintenance) OR ("Document
    Title":trace link maintenance))
OR
(("Abstract":traceability maintenance) OR ("Abstract":trace link
     maintenance))
OR
(("Author Keywords":traceability maintenance) OR ("Author
    Keywords":trace link maintenance)))
AND
((traceability maintenance) OR (trace link maintenance))
```

In addition to the query execution two filters haven been applied. First articles haven been filter by *Publication Type Filter* so that only *Conferences*, *Journals & Magazines* and *Early Access Articles* are included (exclude *Standards*). Second the articles have been filter by date to only include articles with publication dates from the year 2000 and later. This initially resulted in 196 articles received from the IEEE Xplore Digital Library. Evaluating the articles title and conference or journal series limited the results to 65 articles. From these 65 articles 34 articles were also included in the articles received by the ACM Digital Library. Thus 31 new articles have been kept from IEEE Xplore Digital Library.

[2] https://ieeexplore.ieee.org/search/advsearch.jsp?expression-builder

Springer Link

For *Springer Link*[3] the modification of the basic query was not necessary, since the search considers meta-data and article content by default. Thus the key word query of Listing A.1 has been used. In addition to the query execution the articles have been filter by date to only include articles with publication dates from the year 2000 and later. This initially resulted in 34 articles received from Springer Link. Evaluating the articles title and conference or journal series limited the results to 21 articles. From these 21 articles 19 articles were already included in the articles previously received by the ACM Digital Library and IEEE Xplore Digital Library. Thus 2 new articles have been kept from Springer Link.

ScienceDirect (Elsevier Journals)

For *ScienceDirect*[4] the advanced search functionality has been used. The base key word query of Listing A.1 has been used along with the term search field. In addition the results were restricted to be from year 2000 or later. This resulted in 43 publication. Further the refinement functionality with the Article type (Review articles, Research articles and Book Chapters) and Publication title (only Computer Science specific publications, i.e. Journal of Systems and Software (6), Information and Software Technology (2)) refinement have been applied. This resulted in 7 publications. Further evaluation of the publications titles finally resulted in 3 relevant publications. However, only one of these three publications was not already included by the results of the other search engines and the new one found is not a *TM* approach but a set of guidelines on how to set up traceability maintenance in an cooperate environment.

Scopus (Elsevier)

For *Scopus*[5] the search functionality has been used with the key word query from Listing A.1 and a limitation of the search to the title, abstract, keywords and document text. In addition the date range filter has been used to only include publications from and after the year 2000 and the subject area has been limited to computer science and engineering. This results in 52 publications. Further evaluation of the publications titles finally resulted in 6 relevant publications but they were all already included by the results of the other search engines.

A.1.3 Distinct Approach Filtering

Before the result evaluation the found publications haven been composed so that there is only one primary publication for one *TM* approach. The 26 relevant pub-

[3] `https://link.springer.com/advanced-search`
[4] `https://www.sciencedirect.com/search/advanced`
[5] `https://www.scopus.com/`

Table A.1. Trace Link Maintenance Approaches with Multiple Publications

Selected Pub.	Other Publications about the same *TM* Approach	Publications Source and Selection Rationale
P10, Mäder and Gotel [2012a]	• [Mäder et al., 2008a], traceMaintainer tool initial Paper (technical description), • [Mäder et al., 2008b], traceMaintainer tool 2nd Paper, • [Mäder et al., 2008c], traceMaintainer evaluation Paper, • [Mäder et al., 2009], traceMaintainer tool components and architecture description, • [Mäder et al., 2009], traceMaintainer technical details of update process Paper • [Mäder and Gotel, 2012b], book chapter about goal oriented trace link maintenance which uses traceMaintainer as sample tool),	all publications were found by the key-word search, the most actual publications with the most extensive evaluation has been selected
P13, Rahimi et al. [2016]	• [Rahimi, 2016], initial publication • [Rahimi and Cleland-Huang, 2018], journal paper	all publications were found by the key-word search, the publication with the most extensive evaluation has been selected
P14, Schwarz et al. [2010]	• Schwarz [2009], initial publication	more recent publication found by forward snowballing
P15, Seibel et al. [2012]	• Seibel et al. [2010], initial publication	more recent publication found by forward snowballing

lications contained 16 distinct approaches. As shown in Table A.1, 4 of the 16 approaches where described in multiple publications. The right column of Table A.1 shows the rationales why the 4 primary publication haven been selected. In summary the reasons are the novelty of the publication, the comprehensiveness of the actual *TM* approach description and performed evaluations.

A.2 Results

In the following section more detailed descriptions of the reviewed *TM* approaches are provided.

A.2.1 Detailed Description of Trace Link Maintenance Approaches

In this section the 16 *TM* approaches are described with a standardized template comprising an overview of the approach and the linked artefacts, the four steps of the *TM* process and a description of a potentially performed evaluation. This standardized approach descriptions were used to create the evaluation tables and the short approach descriptions in the *TM SLR* result Section 4.3 of Chapter 4.

P01: Establishing and Maintaining Traceability Between Large Aerospace Process Standards – Armbrust et al. [2009]

Linked Artefacts and Approach Overview

The link maintenance approach of Armbrust et al. is completely manual and supports authors to maintain links between sections of different specification documents

during the change of the document sections. The approach uses a table with information about links directly in the sections of the specification documents. The table with link information consists of a list of all linked sections and the change status of the linked section. The change status is indicated by a flag (Tailored (T): modify, Tailored Out (TO): remove, Unchanged (U): copy paste action of exiting unchanged content).

Trace Link Maintenance Process

The manual maintenance of links is performed by a two different authors.

Step 1: Detection of Change

The first author changes a section and after that follows all links to other sections listed in the link information table of the section.

Step 2: Detection of impacted Links and Further Output

In the linked sections the first author adds flags to the link information tables of these linked sections to indicate the performed change. The flag added to each link is the output of detection.

Step 3: Determination of Necessary Link Change

A second author has to change one of the sections, which have a flag indicating a change in one of their linked sections set by the first author. Based on the flag the second author decides whether the link is still relevant or not.

Step 4: Execution of Change

In consequence the second author removes the flag and potentially the link in both link information tables, i.e. the link information table in the section actually edited and in the link information table of the linked section.

Evaluation

Armbrust et al. evaluated their approach qualitatively in experiment with developers consisting of three parts. First a general explanation of the approach was given to the developers. Second the developers used the approach in a previously created application scenario. Third the developers were interviewed about the applicability and usability of the approach. In the paper Armbrust et al. report about the overall positive statements of the interviewed developers about the approach.

P02: A Semi-Automated Approach for the Co-Refinement of Requirements and Architecture Models – Blouin et al. [2017]

Linked Artefacts and Approach Overview

In the approach of Blouin et al. links are maintained between requirements and architectural models. Therefore the approach builds on the concept of co-refinement, i.e. if architecture model elements change linked requirements elements have to change as well and vice versa. For the approach an existing architecture modelling language has been extended to also be capable of modelling requirements. The prototype implementing the approach uses this architecture- and requirements modelling language with an existing modelling tool called story driven modelling tool (SDM). The modelling language in the tool is used to define a co-refinement scheme consisting of architectural refinement rules and rules to maintain links. The architectural refinement rules are used to detect certain architectural refinements. The link maintenance rules are used for automatically performing link changes.

Trace Link Maintenance Process

Step 1: Detection of Change

In the approach links are maintained ongoing. The tool implementing the approach monitors changes in linked artefacts by tracking interactions of users that change the artefacts content.

Step 2: Detection of impacted Links and Further Output

The architectural refinement rules use the model elements (types, attributes) affected by a change and parent- and child relation of model elements to detect certain types of architectural refinements. These detected types of architectural refinement are the output of impact detection.

Step 3: Determination of Necessary Link Change

For the maintenance of links the maintenance rules also contained in the co-refinement scheme are used. These link maintenance rules use the previously detected type of architectural refinement and involved links and artefacts to determine the necessary link change.

Step 4: Execution of Change

If a certain architectural refinement has been detected and there are rules to maintain links for this architectural refinement, links are maintained automatically by executing the link maintenance rules. Since the link maintenance- and architectural refinement rules only cover a limited set of cases, a user notification with the affected artefacts and links is generated, if automatic link maintenance is not possible (i.e. not covered by the rules). In that case the user has to perform the link maintenance manually.

Evaluation

Blouin et al. did not perform a real evaluation. Instead they showed in a proof of concept example, that the approach can be implemented by using story diagrams and the SDM tool for story diagram creation and execution.

P03: Event-based traceability for managing evolutionary change – Cleland-Huang et al. [2003]

Linked Artefacts and Approach Overview

The approach of Cleland-Huang et al. supports the maintenance of links between requirements and source code by providing information on changed artefacts and their links to the original developer. The approach does not use any rules.

Trace Link Maintenance Process

Step 1: Detection of Change

In the approach a tool monitors artefact change events performed by software developers on linked artefacts for ongoing maintenance of the links.

Step 2: Detection of impacted Links and Further Output

If the tool detects a change on a linked artefact, it sends a notification to the owner, i.e. original author, of the artefact. The notification informs the owner that a linked artefact was changed and includes a list of all potentially affected links. All links from the changed artefact to other artefacts are considered as potentially affected.

Step 3: Determination of Necessary Link Change

Based on the received notification the artefact owner decides if link maintenance is necessary.

Step 4: Execution of Change

The artefact owner manually changes links according to the manually determined link change.

Evaluation

No evaluation has been performed. In Section 5 of the paper the authors explain their prototypical approach implementation and demonstrate its feasibility for providing information about changes on linked artefacts to artefact owners.

P04: Breaking the Big-Bang Practice of Traceability: Pushing Timely Trace Recommendations to Project Stakeholders – Cleland-Huang et al. [2012]

Linked Artefacts and Approach Overview

In the approach of Cleland-Huang et al. links between *UML* model elements and requirements are maintained.

The approach uses so called trace obligations which are a set of source and target artefact specific rules. The rules comprised by a trace obligation state whether traceability goals for an artefact are fulfilled, i.e. all required links exist. An example for such a traceability goal is: a requirement must be linked with at least one class in a class diagram.

These rules are implemented in a tool and are executed when a user changes or creates a traceable artefact. The tool then presents a list of artefacts and their links not fulfilling their traceability goals. This list with links and artefacts is called trace recommendations by the authors.

Trace Link Maintenance Process

Step 1: Detection of Change

In the approach changes on artefacts are monitored.

Step 2: Detection of impacted Links and Further Output

Based on the rules of the trace obligations the tool presents trace recommendations. The links and artefacts of the trace recommendation determined by trace obligations are the output of impacted detection.

Step 3: Determination of Necessary Link Change

The developer uses the trace recommendations to manually determine the necessary link change.

Step 4: Execution of Change

The developer performs changes on the recommended artefacts and links manually.

Evaluation

The approach has been evaluated quantitatively by a simulation in the TraceLab simulation environment tool. A gold standard for trace links has been created manually by the authors. Precision and recall values for trace links recommended by the approach have been calculated. In the best cases the precision was 72.00% and a recall was 22.28%.

P05: A State-based Approach to Traceability Maintenance – Drivalos-Matragkas et al. [2010]

Linked Artefacts and Approach Overview

The approach of Drivalos-Matragkas et al. builds on the concept of model driven software engineering in which all used artefacts and trace links between the artefacts are defined in meta-models.

Maintenance specific data attributes of model elements defined in the meta-models and rules associated with these attributes capture whether link maintenance is necessary when certain attributes are changed by a user. An example for such a maintenance attribute is the model element name and an example for a rule associated with the name attribute is *if the name of a model element changes, links and linked other model elements have to be checked.*

Trace Link Maintenance Process

Step 1: Detection of Change

Changes on traced artefacts are monitored by the approach.

Step 2: Detection of impacted Links and Further Output

If a change to a model element occurs, queries to access the model elements maintenance data attributes are executed. The resulting list with model elements and maintenance attributes is passed to the determination of necessary link change step.

Step 3: Determination of Necessary Link Change

The rules of the maintenance attributes for all model elements received from the detection are executed.

Step 4: Execution of Change

For links with unique maintenance options determined by execution of the maintenance attribute specific rules, maintenance is performed automatically. If multiple maintenance options exist, a notification with the link and linked artefacts to be checked manually by a user is generated.

Evaluation

No evaluation has been performed. In the paper the authors demonstrate their approach with an example along with their EMF-based prototypical implementation.

P06: Semi-automatic Establishment and Maintenance of Valid Traceability in Automotive Development Processes – Fockel et al. [2012]

Linked Artefacts and Approach Overview

In the approach of Fockel et al. textual requirements, formulated with a standardized controlled natural language, are linked to model elements. In the approach different model abstraction levels exist.

The approach uses so called triple graph grammars (TGG) for the creation of model elements from the requirements texts. TGGs are a formalism to specify relations between different models. In the approach TGGs are used to define declarative rules using the model element types of linked source and target artefacts.During the creation of model elements from requirements texts these declarative rules are applied and elements of different model levels are linked with each other. To check the validity of created links the approach uses object constrained language (OCL) to define traceability related constrains. An example for such a constrained is *To each requirement defining a logical component at least one function has to be linked.* If linked elements are changed, the constraint check evaluation for links is performed.

Trace Link Maintenance Process

Step 1: Detection of Change

In the approach changes on requirements artefacts are monitored.

Step 2: Detection of impacted Links and Further Output

If a change to an existing requirement or the creation of a new requirement is detected by the approach, constraints to validate the correctness of existing trace links and the missing of trace links are executed. The result of the constraints validation is a list with requirements and model elements that violate the constraints.

Step 3: Determination of Necessary Link Change

The necessary link change is determined manually by a user with the help of the constraint violation list from the impact detection. For this the approach generates a specific notification like *requirement X is missing a link to a logical component* for each constraint violation of a model element or requirement.

Step 4: Execution of Change

A user has to manually update the trace links.

Evaluation

The approach has been evaluated qualitatively by interviewing developers after presenting them the approach. The interviewed developers stated that the presented techniques reduce the manual work.

P07: Supporting Traceability Through Affinity Mining – Gervasi and Zowghi [2014]

Linked Artefacts and Approach Overview

In the approach of Gervasi and Zowghi links are maintained within a requirement specification between high- and low-level requirements.

To detect links to maintain the approach calculates an affinity score for pairs of terms based on the occurrence frequency of pairs of terms in already linked high and low-level requirement documents. The stemmed form of nouns, adjectives, adverbs and verbs are considered as terms. A pair of term consists of one term from a high level document and one term from a low level document. For every occurrence of a pair of terms in already linked high- and low-level documents the affinity score for the pair of terms is increased by one.

In an initial learning phase of the approach the affinity scores for all pair of terms in already linked documents are calculated. The calculated affinity scores are then used together with a configurable threshold by the approach to judge about linking of new documents and changed documents. If the affinity score of two documents is above the specified threshold and no links exist a new link is created. If the affinity score of two documents is below the specified threshold and a link exists the link is removed. The approach can be applied once to validate the linking of an existing project or for the ongoing maintenance of existing links while artefacts are changed by a user.

Trace Link Maintenance Process

Step 1: Detection of Change

Changes on artefacts are monitored by the approach.

Step 2: Detection of impacted Links and Further Output

After a requirement document has been changed, affinity scores for all documents are recalculated. The recalculated affinity scores for the requirements documents are passed to the determination of necessary change step.

Step 3: Determination of Necessary Link Change

To determine if link changes are necessary the approach checks, if the recalculated affinity score for two requirement documents is above or below the specified threshold.

Step 4: Execution of Change

If the recalculated affinity score for linking two requirement documents drops below the specified threshold, an existing link is removed. If the recalculated affinity score for linking two requirement documents rises above the specified threshold a new link is created.

Evaluation

The approach has been evaluated qualitatively by simulating the application of the approach with a prototypical tool and a known set of links (gold standard). In the best case a precision of 85.3% and a recall of 64.4% for maintained links could be achieved.

P08: Code Patterns for Automatically Validating Requirements-to-Code Traces – Ghabi and Egyed [2012]

Linked Artefacts and Approach Overview

In the approach of Ghabi and Egyed links between requirements and source code are maintained automatically. Links to source code can exist on class (complete file) and method (part of a file) level. The approach requires an executable system like a compiled version of the systems source code, the corresponding requirements, and the requirements to code trace links.

The approach uses call relations in source code to determine whether existing links from source code to requirements are valid or whether links are missing. The call relations are determined in the approach by executing the system. The call relations are used investigating the known existing trace links of neighbouring methods, i.e. calling and called methods. The approach assumes that a given method is likely implementing a given requirement if it is called or calls other methods that also implement the given requirement.

Based on this assumption existing links and call relations between source code files, are used by patterns defined in the approach to assign a score (trace expectation) to the methods in the source code. An example for such a pattern is the surrounding pattern. The surrounding pattern defines that if a method A is called by a method B and calls a method C and both methods B and C are connected to the same requirement R by a trace link, the method A should also be connected to that requirement R by a trace link. The overall score of a method is calculated in the approach by executing all patterns for the method. The score for a method is then used to judge the existing links and to determine missing links of this method.

Trace Link Maintenance Process

Step 1: Detection of Change

Artefact change is indicated manually by a user. Therefore the user manually triggers the approach with the required input data, i.e. compiled source code, requirements, requirements to code trace links.

Step 2: Detection of impacted Links and Further Output

The approach evaluates patterns which use the call relations and existing links to assign a numerical score to methods in the code. The score is the output.

Step 3: Determination of Necessary Link Change

The score for the methods in the code is used to judge the existing links and to determine missing links.

Step 4: Execution of Change

If the score for a method is below a certain value, links are removed from the method. If the score fore a method to be linked to a requirement is above a certain value and no link exist, a new links is added.

Evaluation

The approach has been evaluated qualitatively by simulating the application of the approach with a prototypical tool and a known set of links (gold standard) for four projects. First the gold standard links were used and supplemented with wrong links in different manners (randomly, by a user). Then the tool has been used to detect the introduced wrong links. In the best case for precision a precision of 94.1% with a recall of 84.3% and in the best case for recall a recall of 96.1% with a precision of 92.8% could be achieved.

P09: Towards Recovering and Maintaining Trace Links for Model Sketches Across Interactive Displays – Kleffmann et al. [2013]

Linked Artefacts and Approach Overview

In the approach of Kleffmann et al. links between sketches and diagrams on interactive displays are maintained. The approach does not use any rules.

Trace Link Maintenance Process

Step 1: Detection of Change

In the approach a tool observes user interactions with sketches and diagrams on interactive displays.

Step 2: Detection of impacted Links and Further Output

If a linked artefact is changed, the tool highlights the links and the linked artefacts and prompts the user to manually check the linking.

Step 3: Determination of Necessary Link Change

Based on the highlighting of links and artefact affected by an artefact change, the user determines necessary link change manually

Step 4: Execution of Change

The user performs necessary link change manually.

Evaluation

No evaluation has been performed. In the conclusion the authors report about a planned future evaluation.

P12: Rigorous identification and encoding of trace-links in model-driven engineering – Paige et al. [2011]

This approach is related to the approach P05 (cf. Section A.2.1). It uses the same principle of connection model elements defined in different meta-models and extends this principle with a separate meta-model for traceability.

Linked Artefacts and Approach Overview

The approach of Paige et al. enables the creation of links between model elements defined in different meta-models. The approach uses the *Traceability Meta- modelling Language* (TML) to define a meta-model for typed trace links between model elements from the different meta-models. In addition constrains for the trace links are defined by using *Epsilon Validation Language* (EVL). The constraints cover definitions to ensure that trace links are valid, e.g. a certain model element from one meta-model is only allowed to be linked with a certain model element from another meta-model, and that all model elements requiring linkage are linked, e.g. all Class instances from an ObjectOrienetedMetamodel have to be linked to one Actor instance from the IstarMetamodel.

Trace Link Maintenance Process

Step 1: Detection of Change

Artefact change is indicated manually by a user.

Step 2: Detection of impacted Links and Further Output

The constraints to validate the correctness of existing trace links and the missing of trace links are executed. The result of the constraints validation is a list with requirements and model elements that violate the constraints.

Step 3: Determination of Necessary Link Change

The necessary link change is determined manually by a user with the help of the constraint violation list from the impact detection.

Step 4: Execution of Change

Link maintenance is performed manually by a user using the results of the constraints.

Evaluation

No evaluation has been performed. However, the authors illustrated the usage of their approach within two projects.

P10: Towards Automated Traceability Maintenance – Mäder and Gotel [2012a]

Linked Artefacts and Approach Overview

In the approach of Mäder and Gotel trace links between requirements and analysis and design models expressed in *UML* diagrams are maintained. Link maintenance is performed in a tool (*traceMaintainer*) that analyzes change events that have been captured while working within a third-party *UML* modeling tool. In the traceMaintainer tool certain sequences of captured events are comprised as development activities. The development activities are matched with predefined rules that direct the update of impacted trace links. An example for such a development activity is *convert class into component*, which comprises the events *remove class, add new component and rename component*. The rules use the type of *UML* model element, the changed attribute and existing links. The rules state if a link requires maintenance and which kind of maintenance is required (delete, move, etc.).

Trace Link Maintenance Process

Step 1: Detection of Change

The traceMaintainer tool monitors changes of a user performed in a third-party *UML* modelling tool in *UML* diagrams and requirements.

Step 2: Detection of impacted Links and Further Output

All development activities detected by the traceMaintainer tool are passed as output to the next step for further processing.

Step 3: Determination of Necessary Link Change

Based on a previously detected development activity rules are executed by the traceMaintainer tool to determine the necessary link change. For the example of the *convert class into component development activity* the link maintenance rules are *all links previously using the removed class as source or target now have to use the new created component as source or target.*

Step 4: Execution of Change

If a development activity has been detected and there are rules how to maintain links for the detected development activity, link maintenance is performed automatically by execution of the rules within the tool. If no development activity has been detected or there are no link maintenance rules for a detected development activity, the involved artefacts are shown to the user to manually maintain the affected links.

Evaluation

The approach has been evaluated qualitatively and quantitatively together with an industrial partner. Within this evaluation the approach has been applied over one year in two projects. Qualitative feedback has been collect by interviewing developers using the tool and then using the feedback to improve the tool. For the quantitative evaluation the potential reduction in manual effort and the quality of resulting trace links was evaluated. The authors state that the overall reduction of manual effort was 71% compared to a complete manual maintenance of links, i.e. only 29% of trace links had to be maintained manually when using the traceMaintainer tool. For the quality of resulting trace links the authors report a precision of 86% and a recall of 88%.

P11: Capra: A Configurable and Extendable Traceability Management Tool – Maro and Steghöfer [2016]

Linked Artefacts and Approach Overview

The approach of Maro and Steghöfer is implemented in a tool called *Capra*. Capra is an open source traceability management tool implemented as Eclipse plug-in using the Eclipse Modelling Framework (EMF). The tool enables a user to link artefacts managed within the Eclipse *IDE* (e.g. Java Source code) with EMF model elements. For maintenance of existing links the approach comprises a notification mechanism which informs a user to maintain existing links whenever linked artefacts change.

Trace Link Maintenance Process

Step 1: Detection of Change

The Capra tool monitors changes of artefacts.

Step 2: Detection of impacted Links and Further Output

If a changed artefact is linked to another artefact, the tool considers the involved links as potentially impacted. After a user saves a performed change to an artefact, the tool notifies the user to manually maintain involved links. In the notification the tool presents the affected links and artefacts.

Step 3: Determination of Necessary Link Change

Link change is determine manually be a user, using the links and artefacts presented in the notification.

Step 4: Execution of Change

A user performs the changes to links manually.

Evaluation

No evaluation has been performed. In the paper the authors illustrated the usage of their tool along with a running example.

P13: Evolving Requirements-to-Code Trace Links across Versions of a Software System – Rahimi et al. [2016]

Linked Artefacts and Approach Overview

The *Trace Link Evolver* (TLE) is an approach of Rahimi et al. that is capable to maintain trace links between source code (classes or methods) and requirements based on the detection of refactorings between two successive versions of source code.

The TLE approach can be applied retrospectively if two version of source code files exist,t or it can be applied ongoing every time a developer creates a new version of source code files, i.e. the developers performs a commit to a *Version Control System* (*VCS*). TLE defines 24 change scenarios organized into the six high level change categories: add class, delete class, add method, delete method, modify method, and basic. Each of the 24 change scenarios captures a typical refactoring action comprised of several changes to source code performed by a developer.

To define the scenarios heuristic are used. The heuristics are built from 49 rules. These rules check for add, delete and modification of methods and classes, whether links are involved between classes and classes and classes and requirements, for methods in classes, for association between classes and for the existence of classes, methods and requirements. For each scenario there is also a so called link evolution heuristic. This link evolution heuristic is a rule which defines how to maintain trace links involved in a scenario.

The TLE approach is implemented in a tool. For 18 of the 24 scenarios the call graph of the source code and textual similarity calculations, based on *Vector Space Model* (*VSM*), between requirements and source code are used. For the *VSM*-based textual similarity calculation preprocessing techniques stop word removal and stemming have been applied. A threshold score for the calculated cosine similarity was established as half of the highest similarity score between artefacts. Pairs of artefacts (i.e. a source code file and a requirement) with scores above the threshold were considered to be similar and thus linked with each other.

For the other 6 of the 24 scenarios a tool called *Refactoring Crawler*, which directly detects a certain refactoring based on two versions of source code files, is used. For the refactoring detection the tool only requires these two versions of source code files as input. In consequence the detected refactoring, the requirements, the two versions of source code files and existing links for the first version of source code files are used to create the actual links between the second version of source code files and the requirements. An example for a refactoring detected by the Refactoring Crawler tool is Renamed Class. If the rename class refactoring has been detected, existing trace links can become invalid.

Trace Link Maintenance Process

Step 1: Detection of Change

Detection of changes can be performed retrospectively by using two versions of artefacts in a version control system or by monitoring changes of a user. In both cases the difference between two versions of source code files and requirements are used.

Step 2: Detection of impacted Links and Further Output

Impacted links are detected by identifying one of the 24 change scenarios. The detected change scenario is passed as output to the determination of necessary link change step.

Step 3: Determination of Necessary Link Change

The combination of results from the rules of an identified change scenario state if links require maintenance. The necessary link change is determined by the change scenario specific link evolution heuristics.

Step 4: Execution of Change

The Link evolution actions of a change scenario specific link evolution heuristic are executed by the TLE tool.

Evaluation

The approach has been evaluated quantitatively in two setups. In the first initial evaluation setup 13 versions of two small Java programs have been used. The different versions have been generated by developers in controlled 90 minutes modification sessions and the gold standard links have been created by the authors themselves. In the first evaluation a precision of 92.4% and a recall of 96.4% was achieved (compared to a recall of 66,9% and precision of 35.2% when using *VSM*). In the second evaluation setup the authors used the larger *apache Casandra* open source data base project. Features (requirements) have been taken from the projects issue tracker system. 11 source code version haven been taken from the projects *Version Control*

System (*VCS*). Trace links haven been created by the authors using the data from the requirements, e.g. if a class was mentioned in a feature a trace link has been created. The trace links created by the authors have also been supplemented and verified by also creating links with *VSM*. In the second evaluation a precision of 91.9% and a recall of 94.5% was achieved (compared to a recall of 5% or 13% and precision of 10% or 5% when using *VSM* or *LSI* respectively).

P14: Graph-based traceability: a comprehensive approach – Schwarz et al. [2010]

Linked Artefacts and Approach Overview

The approach of Schwarz et al. enables the creation of trace links between model elements. The approach uses *grUML* which is a graph technology extension to *UML*. It enables the representation of any *UML* model element and arbitrary extensions of the *UML* meta-model as graph. In addition *grUML* provides a querying language for such a graph. In the approach of Schwarz et al. *grUML* is used to provide a basic traceability meta-model (traceability information model) for linked entities (*UML* model elements, and model elements based on own extensions of the *UML* meta-model), (typed) trace links and rules. The rules are represented as attributes of linked entities and are used to keep and make trace links consistent. Link maintenance in the approach is implemented within model transformation from source to target models.

Trace Link Maintenance Process

Step 1: Detection of Change

Detection of change is performed by tool based monitoring of change events for model elements.

Step 2: Detection of impacted Links and Further Output

Impacted links are detected by monitoring changes in target models, followed by subsequent changes in source models on the same model element. The detected change type is based on where it was detected (target or source model) and on which kind of element (link element or element). The detected change type is passed to the next processing step as output.

Step 3: Determination of Necessary Link Change

Necessary link change is determined in two ways using the previously detected change type. If a change has been detected in a source model link or linked element it can be transferred automatically to the target model. If a change has been detected in a target model link or linked element or in a target model link or linked element

and source model link or linked element the elements are presented to the user for manual processing.

Step 4: Execution of Change

Changes in target model links and linked elements are automatically processed by the approach other changes are presented in a notification to the user for manual processing.

Evaluation

No evaluation has been performed. The authors report about a planned future evaluation within the research project in which the approach has been developed. Within the paper the authors present their approach along with an running example.

P15: Traceability in Model-Driven Engineering: Efficient and Scalable Traceability Maintenance – Seibel et al. [2012]

Linked Artefacts and Approach Overview

In the approach of Seibel et al. all model elements defined in a meta-model can be linked with each other. The approach uses a meta-model capable of traceability for model elements on different model abstraction levels. The approach defines two kind of trace links fact- and obligation links. Fact links are *normal* links which show that model elements are connected with each other. In addition to fact links, obligation links contain a check mechanism to use information from linked model elements, e.g. such a check could be name of source model element should be the same as the name of the target model element of the link.

The maintenance of these two kinds of links is separated into two processes (steps). First fact links are maintained in the so called localization process and second obligation links in the execution process. For the localization process the approach defines rules based on model element attributes and the connection of links and linked model elements on different abstraction levels. The rules enable the automatic maintenance of links on one abstraction level, if links or linked elements change on another abstraction level. For the execution step tasks are used to maintain obligation links. These tasks consider the data flow between model elements to maintain obligation links.

Link maintenance like creation of missing links and deletion of wrong links can be performed in two ways with the tool. The first way is the batch mode manually triggered by a user (e.g. for the initial application), the second way is the incremental mode which triggers the approach after a change of artefacts.

Trace Link Maintenance Process

Step 1: Detection of Change

In batch mode the link maintenance process of the approach is manually triggered by a user. In incremental mode the approach monitors changes on linked model elements.

Step 2: Detection of impacted Links and Further Output

The creation- and deletion rules of the localization process are executed. Impacted fact links are detected and maintained by the rules. The execution process is executed to maintain obligation links. The result of the execution process is a list with change tasks. Each change tasks specifies how to maintain the obligation links of one model element.

Step 3: Determination of Necessary Link Change

Determination of necessary link change is covered by previous step description.

Step 4: Execution of Change

Fact links are maintained automatically by executing the localization process and by executing the update tasks previously created by the execution process.

Evaluation

The approach has been evaluated quantitatively for its scalability and performance. Both modes of the approach, batch and incremental mode haven been evaluated with projects of different sizes. The used evaluation projects where generated automatically. In batch mode the required performance increased linear with linear growth of project size (number of mode elements). In incremental model the same linear increasing project sizes of the batch mode have been used. In addition a different number of changes (5, 60, 120, 200) to linked model elements has been performed. The result for this evaluation was that the number of model elements has no effect and the number of changes has a small effect on the performance, i.e. more changes resulted in slower execution.

P16: Software Artifacts Management Based on Dataspace – Ying et al. [2009]

Linked Artefacts and Approach Overview

The approach of Ying et al. uses the concept of *dataspaces* to maintain artefacts stored in software repositories and links between those artefacts. A *dataspace* consists of participants which are users with a specific role and relationships and models relations between data repositories and the artefacts in these repositories. A *dataspace* is used to manage the life cycle of software artefacts and holds all information

relevant for a specific concern, i.e. a task performed by a user. To perform automatic or semi-automatic recovery and maintenance of links data space discovery with an ontology also describing traceability between artefacts is used.

First *Natural Language Processing* (*NLP*) and source code analysis are used to extract relevant information from software artefacts. The result of this extraction is a single graph representation of software artefacts and the trace links between the artefacts. Second relations between ontologies used in the approach and the graph representation of software artefacts are created by graph merging, i.e. relations between the nodes in the ontology and the nodes (artefacts) in the software artefact graph are created. Third reasoning is used to infer implicitly trace links among the software artefacts. Implicitly trace links are trace link not already created in the extraction step. The approach can be applied initially if no trace links exist or it can be applied along the changes of artefacts.

Trace Link Maintenance Process

Step 1: Detection of Change

The approach monitors changes in artefacts.

Step 2: Detection of impacted Links and Further Output

In the data extraction step and reasoning step the approach creates links based on information extracted from artefacts and an ontology. Rules based on the results of reasoning with an ontology and data extracted from linked artefacts state if links require maintenance. The output of detection is a list with links including a change type for the link. The change type can indicate to create a new link, to delete or potentially delete a link.

Step 3: Determination of Necessary Link Change

Necessary change is determined by the type of link change for each link from the list of links created as output of the detection step.

Step 4: Execution of Change

New links are created automatically, links to be deleted are deleted automatically and links to potentially be deleted are shown to the user.

Evaluation

No evaluation has been performed.

Bibliography

Orlena Gotel, Jane Cleland-Huang, Jane Huffman Hayes, Andrea Zisman, Alexander Egyed, Paul Grünbacher, Alex Dekhtyar, Giuliano Antoniol, Jonathan Maletic, and Patrick Mäder. Traceability Fundamentals. In *Software and Systems Traceability*, pages 3–22. Springer, 2012a. (cit. on pp. 3, 4).

Jane Cleland-Huang, Orlena Gotel, Jane Huffman Hayes, Patrick Mäder, and Andrea Zisman. Software traceability: trends and future directions. In *Proceedings of the on Future of Software Engineering (FOSE)*, pages 55–69, Hyderabad, India, 2014. ACM. (cit. on pp. 3, 4, 5, 19, 20, 33, 34, 35, 39, 120).

Orlena Gotel, Jane Cleland-Huang, Jane Huffman Hayes, Andrea Zisman, Alexander Egyed, Paul Grünbacher, and Giuliano Antoniol. The Quest for Ubiquity: A Roadmap For Software and Systems Traceability Research. In *Proceedings of the 20th IEEE International Requirements Engineering Conference (RE)*, pages 71–80, Chicago, IL, USA, 2012b. IEEE. (cit. on pp. 3, 5, 19, 23, 24, 25, 33, 34, 35, 95).

Andrea De Lucia, Fausto Fasano, Rocco Oliveto, and Genoveffa Tortora. Fine-grained management of software artefacts: the ADAMS system. *Software: Practice and Experience*, 40(11):1007–1034, 2010. (cit. on p. 3).

Patrick Mäder and Alexander Egyed. Do Software Engineers Benefit from Source Code Navigation with Traceability? - An Experiment in Software Change Management. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 444–447, Lawrence, KS, USA, 2011. IEEE. (cit. on p. 3).

Gabriele Bavota, Luigi Colangelo, Andrea De Lucia, Sabato Fusco, Rocco Oliveto, and Annibale Panichella. TraceME: Traceability Management in Eclipse. In *Proceedings of the 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 642–645. IEEE, 2012. (cit. on p. 3).

Elke Bouillon, Patrick Mäder, and Ilka Philippow. A Survey on Usage Scenarios for Requirements Traceability in Practice. In *Proceedings of the 19th International*

Working Conference - Requirements Engineering: Foundation for Software Quality (REFSQ), volume 7830 of *Lecture Notes in Computer Science (LNCS)*, pages 158–173, Essen, Germany, 2013. Springer. (cit. on p. 3).

Patrick Rempel, Patrick Mäder, Tobias Kuschke, and Jane Cleland-Huang. Mind the gap: assessing the conformance of software traceability to relevant guidelines. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, pages 943–954, New York, NY, USA, 2014. ACM/IEEE. (cit. on p. 3).

Patrick Mäder and Alexander Egyed. Do developers benefit from requirements traceability when evolving and maintaining a software system? *Empirical Software Engineering*, 20(2):413–441, 2015. (cit. on pp. 3, 4).

Markus Borg, Per Runeson, and Anders Ardö. Recovering from a decade: a systematic mapping of information retrieval approaches to software traceability. *Empirical Software Engineering*, 19(6):1565–1616, 2014. (cit. on pp. 3, 4, 6, 20, 21, 23, 25, 26, 33, 34, 35, 101, 115, 120).

Annibale Panichella, Andrea De Lucia, and Andy Zaidman. Adaptive User Feedback for IR-Based Traceability Recovery. In *Proceedings of the 8th IEEE/ACM International Symposium on Software and Systems Traceability (SST@ICSE)*, pages 15–21, Florence, Italy, 2015. IEEE. (cit. on p. 3).

Collin McMillan, Denys Poshyvanyk, and Meghan Revelle. Combining Textual and Structural Analysis of Software Artifacts for Traceability Link Recovery. In *Proceedings of the ICSE Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE@ICSE)*, pages 41–48, Vancouver, BC, Canada, 2009. IEEE. (cit. on p. 4).

Andrea De Lucia, Rocco Oliveto, and Genoveffa Tortora. IR-Based Traceability Recovery Processes: An Empirical Comparison of "One-Shot" and Incremental Processes. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 39–48, L'Aquila, Italy, 2008. IEEE. (cit. on p. 4).

Lionel Briand, Davide Falessi, Shiva Nejati, Mehrdad Sabetzadeh, and Tao Yue. Traceability and SysML Design Slices to Support Safety Inspections. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 23(1):1–43, 2014. (cit. on p. 4).

Nan Niu, Tanmay Bhowmik, Hui Liu, and Zhendong Niu. Traceability-Enabled Refactoring for Managing Just-In-Time Requirements. In *Proceedings of the 22nd IEEE International Requirements Engineering Conference (RE)*, pages 133–142, Karlskrona, Sweden, 2014. IEEE. (cit. on pp. 4, 5).

Walid Maalej, Zijad Kurtanovic, and Alexander Felfernig. What Stakeholders Need to Know About Requirements. In *Proceedings of the 4th IEEE International Workshop on Empirical Requirements Engineering (EmpiRE)*, pages 64–71, Karlskrona, Sweden, 2014a. IEEE. (cit. on pp. 4, 11).

Neil A. Ernst and Gail C. Murphy. Case studies in just-in-time requirements analysis. In *Proceedings of the 2nd IEEE International Workshop on Empirical Requirements Engineering (EmpiRE)*, pages 25–32, Chicago, IL, USA, 2012. IEEE. (cit. on p. 4).

Thorsten Merten, Matúš Falisy, Paul Hübner, Thomas Quirchmayr, Simone Bürsner, and Barbara Paech. Software Feature Request Detection in Issue Tracking Systems. In *Proceedings of the 24th IEEE International Requirements Engineering Conference (RE)*, pages 166–175, Beijing, China, 2016a. IEEE. (cit. on pp. 4, 11, 12, 25).

Thorsten Merten, Daniel Krämer, Bastian Mager, Paul Schell, Simone Bürsner, and Barbara Paech. Do Information Retrieval Algorithms for Automated Traceability Perform Effectively on Issue Tracking System Data? In *Proceedings of the 22nd International Working Conference - Requirements Engineering: Foundation for Software Quality (REFSQ)*, volume 9619 of *Lecture Notes in Computer Science (LNCS)*, pages 45–62, Gothenburg, Sweden, 2016b. Springer. (cit. on pp. 4, 25, 27, 95, 115, 120).

Rebekka Wohlrab, Jan-Philipp Steghöfer, Eric Knauss, Salome Maro, and Anthony Anjorin. Collaborative Traceability Management: Challenges and Opportunities. In *Proceedings of the 24th IEEE International Requirements Engineering Conference (RE)*, pages 216–225, Beijing, China, 2016. IEEE. (cit. on pp. 4, 39).

Salome Maro, Anthony Anjorin, Rebekka Wohlrab, and Jan-Philipp Steghöfer. Traceability maintenance: Factors and guidelines. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 414–425, Singapore, 2016. ACM. (cit. on pp. 4, 39).

Thomas Fritz, Gail C. Murphy, Emerson Murphy-Hill, Jingwen Ou, and Emily Hill. Degree-of-Knowledge: Modeling a Developer's Knowledge of Code. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 23(2):1–42, 2014. (cit. on p. 6).

Martin Konôpka and Mária Bieliková. Software Developer Activity as a Source for Identifying Hidden Source Code Dependencies. In *Proceedings of SOFSEM: Theory and Practice of Computer Science - 41st International Conference on Current Trends in Theory and Practice of Computer Science*, volume 8939 of *Lecture Notes*

in Computer Science (LNCS), pages 449–462. Springer, Pec pod Sněžkou, Czech Republic, 2015. (cit. on pp. 6, 17, 19).

Zéphyrin Soh, Foutse Khomh, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. Noise in Mylyn interaction traces and its impact on developers and recommendation systems. *Empirical Software Engineering*, 23(2):645–692, 2018. (cit. on pp. 6, 17, 57, 76).

Roel J. Wieringa. *Design Science Methodology for Information Systems and Software Engineering*. Springer, Berlin, Heidelberg, 2014. (cit. on p. 7).

Roel J. Wieringa and Ayşe Moralı. Technical Action Research as a Validation Method in Information Systems Design Science. In *Design Science Research in Information Systems. Advances in Theory and Practice*, pages 220–238. Springer, Berlin, Heidelberg, 2012. (cit. on p. 7).

Paul Hübner. Quality Improvements for Trace Links between Source Code and Requirements. In *Proceedings of the REFSQ Workshops, Doctoral Symposium, Research Method Track, and Poster Track*, volume 1564, Gothenburg, Sweden, 2016. CEUR-WS. (cit. on p. 10).

Paul Hübner and Barbara Paech. Using Interaction Data for Continuous Creation of Trace Links Between Source Code and Requirements in Issue Tracking Systems. In *Proceedings of the 23rd International Working Conference - Requirements Engineering: Foundation for Software Quality (REFSQ)*, volume 10153 of *Lecture Notes in Computer Science (LNCS)*, pages 291–307, Essen, Germany, 2017. Springer. (cit. on p. 10).

Paul Hübner and Barbara Paech. Evaluation of Techniques to Detect Wrong Interaction Based Trace Links. In *Proceedings of the 24th International Working Conference - Requirements Engineering: Foundation for Software Quality (REFSQ)*, volume 10753 of *Lecture Notes in Computer Science (LNCS)*, pages 75–91, Utrecht, The Netherlands, 2018. Springer. (cit. on p. 10).

Paul Hübner and Barbara Paech. Increasing Precision of Automatically Generated Trace Links. In *Proceedings of the 25th International Working Conference - Requirements Engineering: Foundation for Software Quality (REFSQ)*, volume 11412 of *Lecture Notes in Computer Science (LNCS)*, pages 73–89, Essen, Germany, 2019. Springer. (cit. on p. 10).

Paul Hübner and Barbara Paech. Interaction-based Creation and Maintenance of Continuously Usable Trace Links. *Empirical Software Engineering*, 2020. (cit. on p. 10).

Jane Huffman Hayes, Alex Dekhtyar, and Senthil Karthikeyan Sundaram. Advancing candidate link generation for requirements tracing: The study of methods. *IEEE Transactions on Software Engineering (TSE)*, 32(1):4–19, 2006. (cit. on pp. 12, 24, 27, 103, 115).

Scott Chacon and Ben Straub. *Pro Git*. Apress L.P., Berkeley, CA, USA, second edition, 2014. (cit. on p. 14).

Hongyu Kuang, Patrick Mäder, Hao Hu, Achraf Ghabi, LiGuo Huang, Jian Lü, and Alexander Egyed. Can method data dependencies support the assessment of traceability between requirements and source code? *Journal of Software: Evolution and Process*, 27(11):838–866, 2015. (cit. on p. 14).

Hongyu Kuang, Jia Nie, Hao Hu, Patrick Rempel, Jian Lu, Alexander Egyed, and Patrick Mäder. Analyzing closeness of code dependencies for improving IR-based Traceability Recovery. In *Proceedings of the 24th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 68–78, Klagenfurt, Austria, 2017. IEEE. (cit. on p. 14).

Achraf Ghabi and Alexander Egyed. Code Patterns for Automatically Validating Requirements-to-Code Traces. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 200–209, Essen, Germany, 2012. ACM. (cit. on pp. 14, 34, 35, 36, 44, 45, 46, 50, 69, 70, 71, 125, 126, 143).

Mona Rahimi, William Goss, and Jane Cleland-Huang. Evolving Requirements-to-Code Trace Links across Versions of a Software System. In *Proceedings of the 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 99–109, Raleigh, NC, USA, 2016. IEEE. (cit. on pp. 14, 44, 45, 47, 48, 50, 69, 70, 71, 125, 126, 135, 148).

Gail C. Murphy, Mik Kersten, and Leah Findlater. How are Java Software Developers Using the Elipse IDE? *IEEE Software*, 23(4):76–83, 2006. (cit. on p. 16).

Walid Maalej, Thomas Fritz, and Romain Robbes. Collecting and Processing Interaction Data for Recommendation Systems. In *Recommendation Systems in Software Engineering*, pages 173–197. Springer Berlin, Heidelberg, 2014b. (cit. on p. 16).

Walid Maalej and Mathias Ellmann. On the Similarity of Task Contexts. In *Proceedings of the 2nd IEEE/ACM International Workshop on Context for Software Development (CSD)*, pages 8–12, Florence, Italy, 2015. IEEE. (cit. on pp. 16, 76).

Jane Cleland-Huang, Patrick Mäder, Mehdi Mirakhorli, and Sorawit Amornborvornwong. Breaking the Big-Bang Practice of Traceability: Pushing Timely Trace Recommendations to Project Stakeholders. In *Proceedings of the 20th IEEE International Requirements Engineering Conference (RE)*, pages 231–240, Chicago, IL, USA, 2012. IEEE. (cit. on pp. 17, 44, 45, 46, 50, 139).

Inah Omoronyia, Guttorm Sindre, Marc Roper, John Ferguson, and Murray Wood. Use Case to Source Code Traceability: The Developer Navigation View Point. In *Proceedings of the 17th IEEE International Requirements Engineering Conference (RE)*, pages 237–242, Atlanta, GA, USA, 2009. IEEE. (cit. on p. 17).

Inah Omoronyia, Guttorm Sindre, and Tor Stålhane. Exploring a Bayesian and linear approach to requirements traceability. *Information and Software Technology*, 53(8):851–871, 2011. (cit. on pp. 17, 19).

Martin Konôpka and Pavol Navrat. Untangling Development Tasks with Software Developer's Activity. In *Proceedings of the 2nd IEEE/ACM International Workshop on Context for Software Development (CSD)*, pages 13–14, Florence, Italy, 2015. IEEE. (cit. on pp. 17, 76).

Mohammad El-Ramly and Eleni Stroulia. Mining Software Usage Data. In *Proceedings of the 1st International Workshop on Mining Software Repositories (MSR@ICSE)*, pages 64–68, Edinburgh, Scotland, UK, 2004. ACM/IEEE. (cit. on p. 17).

Kevin Schneider, Carl Gutwin, Reagan Penner, and David Paquette. Mining a software developer's local interaction history. In *Proceedings of the 1st International Workshop on Mining Software Repositories (MSR@ICSE)*, pages 106–110, Edinburgh, Scotland, UK, 2004. ACM/IEEE. (cit. on p. 17).

Chris Parnin, Carsten Görg, and Spencer Rugaber. Enriching Revision History with Interactions. In *Proceedings of the 3rd International Workshop on Mining Software Repositories (MSR@ICSE)*, pages 155–158, Shanghai, China, 2006. ACM/IEEE. (cit. on p. 18).

Sarah Rastkar and Gail C. Murphy. On what basis to recommend: Changesets or interactions? In *Proceedings of the 6th International Working Conference on Mining Software Repositories (MSR@ICSE)*, pages 155–158, Vancouver, BC, Canada, 2009. ACM/IEEE. (cit. on p. 18).

Romain Robbes and David Rothlisberger. Using developer interaction data to compare expertise metrics. In *Proceedings of the 10th International Working Conference on Mining Software Repositories (MSR@ICSE)*, pages 297–300, San Francisco, CA, USA, 2013. ACM/IEEE. (cit. on p. 18).

Motahareh Bahrami Zanjani, George Swartzendruber, and Huzefa Kagdi. Impact analysis of change requests on source code based on interaction and commit histories. In *Proceedings of the 11th International Working Conference on Mining Software Repositories (MSR@ICSE)*, pages 162–171, Hyderabad, India, 2014. ACM/IEEE. (cit. on p. 18).

Motahareh Bahrami Zanjani, Huzefa Kagdi, and Christian Bird. Using Developer-Interaction Trails to Triage Change Requests. In *Proceedings of the 12th International Working Conference on Mining Software Repositories (MSR@ICSE)*, pages 88–98, Florence, Italy, 2015. ACM/IEEE. (cit. on p. 18).

Andrea De Lucia, Andrian Marcus, Rocco Oliveto, and Denys Poshyvanyk. Information Retrieval Methods for Automated Traceability Recovery. In *Software and Systems Traceability*, pages 71–98. Springer, London, 2011a. (cit. on pp. 19, 20, 22, 23, 24).

Mona Rahimi and Jane Cleland-Huang. Evolving software trace links between requirements and source code. *Empirical Software Engineering*, 23(4):2198–2231, 2018. (cit. on pp. 20, 135).

Michael Rath, Jacob Rendall, Jin L. C. Guo, Jane Cleland-Huang, and Patrick Mäder. Traceability in the Wild: Automatically Augmenting Incomplete Trace Links. In *Proceedings of the 40th International Conference on Software Engineering (ICSE)*, pages 834–845, Gothenburg, Sweden, 2018. ACM/IEEE. (cit. on pp. 20, 25, 116).

Ricardo Baeza-Yates and Berthier de Araújo Neto Ribeiro. *Modern Information Retrieval - the concepts and technology behind search.* Pearson Addison-Wesley, Harlow, England, second edition, 2011. (cit. on pp. 20, 21, 22, 23, 24, 101).

Giuliano Antoniol, Gerardo Canfora, Gerardo Casazza, Andrea De Lucia, and Ettore Merlo. Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering (TSE)*, 28(10):970–983, 2002. (cit. on pp. 20, 34, 35).

Alexander Dekhtyar, Jane Huffman Hayes, and Tim Menzies. Text is Software Too. In *Proceedings of the 1st International Workshop on Mining Software Repositories (MSR@ICSE)*, pages 22–26, Edinburgh, Scotland, UK, 2004. ACM/IEEE. (cit. on p. 20).

Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval.* Cambridge University Press, Cambridge UK, first edition, 2008. (cit. on p. 21).

Andrea De Lucia, Massimiliano Di Penta, and Rocco Oliveto. Improving Source Code Lexicon via Traceability and Information Retrieval. *IEEE Transactions on Software Engineering (TSE)*, 37(2):205–227, 2011b. (cit. on p. 21).

Nasir Ali, Yann-Gaël Gueheneuc, and Giuliano Antoniol. Requirements Traceability for Object Oriented Systems by Partitioning Source Code. In *Proceedings of the 18th Working Conference on Reverse Engineering (WCRE)*, pages 45–54, Limerick, Ireland, 2011. IEEE. (cit. on p. 21).

Jane Cleland-Huang, Brian Berenbach, Stephen Clark, Raffaella Settimi, and Eli Romanova. Best Practices for Automated Traceability. *IEEE Computer*, 40(6): 27–35, 2007. (cit. on pp. 23, 95).

Scott Deerwester, Susan T. Dumais, George W. Furnas, Thomas K. Landauer, and Richard Harshman. Indexing by latent semantic analysis. *Journal of the American Society for Information Science*, 41(6):391–407, 1990. (cit. on p. 24).

Nan Niu and Anas Mahmoud. Enhancing Candidate Link Generation for Requirements Tracing: The Cluster Hypothesis Revisited. In *Proceedings of the 20th IEEE International Requirements Engineering Conference (RE)*, pages 81–90, Chicago, IL, USA, 2012. IEEE. (cit. on p. 24).

Davide Falessi, Massimiliano Di Penta, Gerardo Canfora, and Giovanni Cantone. Estimating the number of remaining links in traceability recovery. *Empirical Software Engineering*, 22(3):996–1027, 2017. (cit. on p. 24).

Ian H. Witten, Eibe Frank, Mark A. Hall, and Christopher J. Pal. *Data Mining: Practical Machine Learning Tools and Techniques*. Elsevier, fourth edition, 2016. (cit. on p. 24).

Christian Bird, Peter C. Rigby, Earl T. Barr, David J. Hamilton, Daniel M. Germán, and Premkumar T. Devanbu. The promises and perils of mining git. In *Proceedings of the 6th International Working Conference on Mining Software Repositories (MSR@ICSE)*, pages 1–10, Vancouver, BC, Canada, 2009. ACM/IEEE. (cit. on p. 25).

Michael Rath, Patrick Rempel, and Patrick Mäder. The IlmSeven Dataset. In *Proceedings of the 25th IEEE International Requirements Engineering Conference (RE)*, pages 516–519, Lisbon, Portugal, 2017. IEEE. (cit. on pp. 25, 59, 82, 110, 119).

Andrea De Lucia, Fausto Fasano, Rocco Oliveto, and Genoveffa Tortora. Recovering Traceability Links in Software Artifact Management Systems using Information Retrieval Methods. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 16(4):1–50, 2007. (cit. on pp. 26, 27, 34, 35, 36, 37, 76, 93, 95, 120).

Martin Frické. Measuring recall. *Journal of Information Science*, 24(6):409–417, 1998. (cit. on p. 27).

B.T. Sampath Kumar and Jyoti Prakash. Precision and Relative Recall of Search Engines: A comparative study of Google and Yahoo. *Singapore Journal of Library and Information Management*, 38(1):124–137, 2009. (cit. on p. 27).

Nasir Ali, Yann-Gael Gueheneuc, and Giuliano Antoniol. Trustrace: Mining Software Repositories to Improve the Accuracy of Requirement Traceability Links. *IEEE Transactions on Software Engineering (TSE)*, 39(5):725–741, 2013. (cit. on pp. 27, 95).

Barbara A. Kitchenham and Stuart Charters. Guidelines for performing Systematic Literature Reviews in Software Engineering. Technical Report Version 2.3, EBSE-2007-01, Software Engineering Group, School of Computer Science and Mathematics, Keele University, Keele, Staffs, ST5 5BG, UK and Department of Computer Science, University of Durham, Durham, UK, 2007. (cit. on pp. 31, 32, 39, 40).

Sunil Nair, Jose Luis De La Vara, and Sagar Sen. A Review of Traceability Research at the Requirements Engineering Conference$^{RE@21}$. In *Proceedings of the 21st IEEE International Requirements Engineering Conference (RE)*, pages 222–229, Rio de Janeiro, RJ, Brazil, 2013. IEEE. (cit. on p. 33).

Yoëlle S. Maarek, Daniel M. Berry, and Gail E. Kaiser. An Information Retrieval Approach For Automatically Constructing Software Libraries. *IEEE Transactions on Software Engineering (TSE)*, 17(8):800–813, 1991. (cit. on pp. 34, 35).

Andrian Marcus and Jonathan I. Maletic. Recovering Documentation-to-Source-Code Traceability Links using Latent Semantic Indexing. In *Proceedings of the 25th International Conference on Software Engineering (ICSE)*, pages 125–135, Portland, OR, USA, 2003. ACM/IEEE. (cit. on pp. 34, 35).

Andrea De Lucia, Fausto Fasano, Rocco Oliveto, and Genoveffa Tortora. Enhancing an Artefact Management System with Traceability Recovery Features. In *Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM)*, pages 306–315, Chicago, IL, USA, 2004. IEEE. (cit. on pp. 34, 35, 37, 93).

Jane Cleland-Huang, Raffaella Settimi, Oussama BenKhadra, Eugenia Berezhanskaya, and Selvia Christina. Goal-centric Traceability for Managing Non-functional Requirements. In *Proceedings of the 27th International Conference on Software Engineering (ICSE)*, pages 362–371, St. Louis, MO, USA, 2005. ACM/IEEE. (cit. on pp. 34, 35, 36).

Leonardo Gresta Paulino Murta, André van der Hoek, and Cláudia Maria Lima Werner. ArchTrace: Policy-Based Support for Managing Evolving Architecture-to-Implementation Traceability Links. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 135–144, Tokyo, Japan, 2006. IEEE. (cit. on pp. 34, 35, 36).

Xuchang Zou, Raffaella Settimi, and Jane Cleland-Huang. Phrasing in Dynamic Requirements Trace Retrieval. In *Proceedings of the 30th Annual International Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 265–272, Chicago, IL, USA, 2006. IEEE. (cit. on pp. 34, 35, 36).

Malcom Gethers, Rocco Oliveto, Denys Poshyvanyk, and Andrea De Lucia. On Integrating Orthogonal Information Retrieval Methods to Improve Traceability Recovery. In *Proceedings of the 27th IEEE International Conference on Software Maintenance (ICSM)*, pages 133–142, Williamsburg, VA, USA, 2011. IEEE. (cit. on pp. 34, 35, 36).

Alexander Delater and Barbara Paech. Analyzing the Tracing of Requirements and Source Code during Software Development. In *Proceedings of the 19th International Working Conference - Requirements Engineering: Foundation for Software Quality (REFSQ)*, volume 7830 of *Lecture Notes in Computer Science (LNCS)*, pages 308–314, Essen, Germany, 2013. Springer. (cit. on pp. 34, 35, 36, 37, 38, 106).

Bogdan Dit, Meghan Revelle, and Denys Poshyvanyk. Integrating information retrieval, execution and link analysis algorithms to improve feature location in software. *Empirical Software Engineering*, 18(2):277–309, 2013. (cit. on pp. 34, 35, 37).

Ove Armbrust, Alexis Ocampo, Jurgen Munch, Masafumi Katahira, Yumi Koishi, and Yuko Miyamoto. Establishing and Maintaining Traceability Between Large Aerospace Process Standards. In *Proceedings of the ICSE Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE@ICSE)*, pages 36–40, Vancouver, BC, Canada, 2009. IEEE. (cit. on pp. 44, 45, 50, 135, 136).

Dominique Blouin, Matthias Barkowski, Melanie Schneider, Holger Giese, Johannes Dyck, Etienne Borde, Dalila Tamzalit, and Joost Noppen. A Semi-Automated Approach for the Co-Refinement of Requirements and Architecture Models. In *Proceedings of the 25th IEEE International Requirements Engineering Conference Workshops (REW)*, pages 36–45, Lisbon, Portugal, 2017. IEEE. (cit. on pp. 44, 45, 46, 50, 137, 138).

Jane Cleland-Huang, Carl K. Chang, and Mark Christensen. Event-based traceability for managing evolutionary change. *IEEE Transactions on Software En-*

gineering (TSE), 29(9):796–810, 2003. (cit. on pp. 44, 45, 46, 47, 50, 69, 70, 138).

Nikolaos Drivalos-Matragkas, Dimitrios S. Kolovos, Richard F. Paige, and Kiran J. Fernandes. A State-based Approach to Traceability Maintenance. In *Proceedings of the 6th ECMFA Traceability Workshop (ECMFA-TW)*, ECMFA-TW '10, pages 23–30, Paris, France, 2010. ACM. (cit. on pp. 44, 45, 46, 47, 50, 140, 145).

Markus Fockel, Jörg Holtmann, and Jan Meyer. Semi-automatic Establishment and Maintenance of Valid Traceability in Automotive Development Processes. In *Proceedings of the 2nd International Workshop Software Engineering for Embedded Systems (SEES)*, pages 37–43, Zurich, Switzerland, 2012. IEEE. (cit. on pp. 44, 45, 46, 50, 141).

Vincenzo Gervasi and Didar Zowghi. Supporting Traceability Through Affinity Mining. In *Proceedings of the 22nd IEEE International Requirements Engineering Conference (RE)*, pages 143–152, Karlskrona, Sweden, 2014. IEEE. (cit. on pp. 44, 45, 46, 50, 142).

Markus Kleffmann, Matthias Book, and Volker Gruhn. Towards Recovering and Maintaining Trace Links for Model Sketches Across Interactive Displays. In *Proceedings of the 7th International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE@ICSE)*, pages 23–29, San Francisco, CA, USA, 2013. IEEE. (cit. on pp. 44, 45, 47, 50, 144).

Patrick Mäder and Orlena Gotel. Towards Automated Traceability Maintenance. *Journal of Systems and Software*, 85(10):2205–2227, 2012a. (cit. on pp. 44, 45, 47, 50, 135, 146).

Salome Maro and Jan-Philipp Steghöfer. Capra: A Configurable and Extendable Traceability Management Tool. In *Proceedings of the 24th IEEE International Requirements Engineering Conference (RE)*, pages 407–408, Beijing, China, 2016. IEEE. (cit. on pp. 44, 45, 47, 50, 147).

Richard F. Paige, Nikolaos Drivalos, Dimitrios S. Kolovos, Kiran J. Fernandes, Christopher Power, Goran K. Olsen, and Steffen Zschaler. Rigorous identification and encoding of trace-links in model-driven engineering. *Software and Systems Modeling*, 10(4):469, 2011. (cit. on pp. 44, 45, 47, 50, 145).

Hannes Schwarz, Jürgen Ebert, and Andreas Winter. Graph-based traceability: a comprehensive approach. *Software and Systems Modeling*, 9(4):473–492, 2010. (cit. on pp. 44, 45, 47, 50, 135, 150).

Andreas Seibel, Regina Hebig, and Holger Giese. Traceability in Model-Driven Engineering: Efficient and Scalable Traceability Maintenance. In *Software and*

Systems Traceability, pages 215–240. Springer, 2012. (cit. on pp. 44, 45, 47, 50, 135, 151).

Pan Ying, Tang Yong, and Ye Xiaoping. Software Artifacts Management Based on Dataspace. In *Proceedings of the WASE International Conference on Information Engineering*, volume 2 of *ICIE '09*, pages 214–217, Taiyuan, Chanxi, China, 2009. IEEE. (cit. on pp. 44, 45, 48, 50, 152).

Mik Kersten and Gail C. Murphy. Using Task Context to Improve Programmer Productivity. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, volume 14, pages 1–11, Portland, OR, USA, 2006. ACM. (cit. on pp. 57, 59, 75).

Kim Herzig and Andreas Zeller. The Impact of Tangled Code Changes. In *Proceedings of the 10th International Working Conference on Mining Software Repositories (MSR@ICSE)*, pages 121–130, San Francisco, CA, USA, 2013. ACM/IEEE. (cit. on pp. 59, 115).

Hiroyuki Kirinuki, Yoshiki Higo, Keisuke Hotta, and Shinji Kusumoto. Hey! Are You Committing Tangled Changes? In *Proceedings of the 22nd International Conference on Program Comprehension (ICPC)*, pages 262–265, Hyderabad, India, 2014. ACM. (cit. on pp. 59, 115).

Ken Schwaber and Mike Beedle. *Agile Software Development with Scrum*. Prentice Hall, Upper Saddle River, NJ, USA, first edition, 2001. (cit. on p. 76).

Marcus Seiler and Barbara Paech. Using Tags to Support Feature Management Across Issue Tracking Systems and Version Control Systems. In *Proceedings of the 23rd International Working Conference - Requirements Engineering: Foundation for Software Quality (REFSQ)*, volume 10153 of *Lecture Notes in Computer Science (LNCS)*, pages 174–180, Essen, Germany, 2017. Springer. (cit. on p. 77).

Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14(2):131–164, 2008. (cit. on p. 117).

Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in Software Engineering*. Springer, Berlin, Heidelberg, first edition, 2012. (cit. on p. 117).

Robert K. Yin. *Case Study Research and Applications: Design and Methods*. Sage Publications Ltd., sixth edition, 2018. (cit. on p. 117).

Patrick Mäder, Orlena Gotel, and Ilka Philippow. Enabling Automated Traceability Maintenance by Recognizing Development Activities Applied to Models. In *Pro-*

ceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE), pages 49–58, L'Aquila, Italy, 2008a. IEEE. (cit. on p. 135).

Patrick Mäder, Orlena Gotel, Tobias Kuschke, and Ilka Philippow. traceMaintainer – Automated Traceability Maintenance. In *Proceedings of the 16th IEEE International Requirements Engineering Conference (RE)*, pages 329–330, Barcelona, CT, Spain, 2008b. IEEE. (cit. on p. 135).

Patrick Mäder, Orlena Gotel, and Ilka Philippow. Rule-Based Maintenance of Post-Requirements Traceability Relations. In *Proceedings of the 16th IEEE International Requirements Engineering Conference (RE)*, pages 23–32, Barcelona, CT, Spain, 2008c. IEEE. (cit. on p. 135).

Patrick Mäder, Orlena Gotel, and Ilka Philippow. Semi-automated Traceability Maintenance: An Architectural Overview of traceMaintainer. In *Proceedings of the 31st International Conference on Software Engineering, Companion Volume (ICSE-C)*, pages 425–426, Vancouver, Canada, 2009. IEEE. (cit. on p. 135).

Patrick Mäder, Orlena Gotel, and Ilka Philippow. Getting Back to Basics: Promoting the use of a Traceability Information Model in Practice. In *Proceedings of the 5th International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE@ICSE)*, pages 21–25, Vancouver, BC, Canada, 2009. IEEE. (cit. on p. 135).

Patrick Mäder and Orlena Gotel. Ready-to-Use Traceability on Evolving Projects. In *Software and Systems Traceability*, pages 173–194. Springer, 2012b. (cit. on p. 135).

Mona Rahimi. Trace Link Evolution across Multiple Software Versions in Safety-Critical Systems. In *Proceedings of the 38th International Conference on Software Engineering, Companion Volume (ICSE-C)*, pages 871–874, Austin, TX, USA, 2016. ACM. (cit. on p. 135).

Hannes Schwarz. Towards a Comprehensive Traceability Approach in the Context of Software Maintenance. In *Proceedings of the 13th European Conference on Software Maintenance and Reengineering (CSMR), Architecture-Centric Maintenance of Large-SCale Software Systems*, pages 339–342, Kaiserslautern, Germany, 2009. IEEE. (cit. on p. 135).

Andreas Seibel, Stefan Neumann, and Holger Giese. Dynamic hierarchical mega models: comprehensive traceability and its efficient maintenance. *Software and Systems Modeling*, 9(4):493, 2010. (cit. on p. 135).