



Technisch-Naturwissenschaftliche
Fakultät

Automatic Approach to Validating Requirement-to-Code Traces

MASTERARBEIT

zur Erlangung des akademischen Grades

Diplomingenieur

im Masterstudium

Informatik

Eingereicht von:

Achraf Ghabi, Bakk. techn.

Angefertigt am:

Institute for Systems Engineering and Automation (SEA)

Beurteilung:

Prof. Dr. Alexander Egyed

Linz, Juli 2010

Abstract

Model traceability is a research field dedicated to establishing and maintaining the mapping between requirements, software models, documentation, and code. It is considered as a sign for process maturity and products quality. And, it is mandated by many standards, such as: *NASA Software Assurance Standard* and *U.K. Office of Government Commerce*.

Traces between requirements and code (requirement-to-code) reveal where requirements are implemented. Such traces are essential for code understanding and change management. Unfortunately, the handling of traces is highly error prone, on one side due to the informal nature of requirements and on another due to the continuous evolution of the code. Incorrect traces are not only less useful for a stakeholder but also misleading in many cases. The correctness of traces is crucial to exploit their usefulness.

This thesis introduces a novel method for validating requirements-to-code traces by considering the calling relationships within the source code. As input, the approach requires existing requirements-to-code traces and as output it identifies potential errors in the input. The approach does this by investigating patterns of traces together with patterns of calling relationships. Our observation is that code that traces to a particular requirement is connected through calling relationships and we exploit this connectivity for validating traceability. The empirical evaluation on four case study systems covering 150 KLOC and 59 sample requirements demonstrates that our approach detects most errors with $> 90\%$ correctness and the quality of validated traces decreases very slowly with less input quality. Our approach is fully automated, tool supported, and efficient (linear computational complexity with input quantity with validation times < 3 seconds on the largest case study).

Kurzfassung

Modell Traceability ist ein Forschungsgebiet gewidmet dem Aufbau und der Pflege von Beziehungen zwischen Anforderungen, Software-Modellen, Dokumentation und Code. Sie wird oft als Maßstab der Prozessreife und Qualität von Software Produkten herangezogen. Sie wird sogar von vielen Standards, wie z.B. dem *NASA Software Assurance Standard* oder *U.K. der Office of Government Commerce*, vorgeschrieben.

Traces zwischen Anforderungen und Code (Requirement-to-Code) zeigen, wo die Anforderungen umgesetzt werden. Solche Traces sind essentiell für das Code-Verständnis und Änderungs-Management. Leider ist der Umgang mit Traces sehr fehleranfällig, auf der einen Seite durch die informellen Eigenschaften der Anforderungen und auf der anderen Seite durch die kontinuierliche Weiterentwicklung vom Code. Falsche Traces sind nicht nur weniger nützlich für Projektbeteiligten, sondern auch irreführend in vielen Fällen. Die Richtigkeit der Traces ist entscheidend um ihr volles Potenzial auszunutzen.

Diese Arbeit stellt ein neuartiges Verfahren zur Validierung von Requirement-to-Code Traces vor, indem man die Aufrufbeziehungen im Quellcode analysiert. Als Eingabe benötigt der Ansatz bestehende Requirement-to-Code Traces und als Ausgabe identifiziert er mögliche falsche Traces in der Eingabe. Dazu, untersucht unser Ansatz Muster von Traces zusammen mit Aufrufbeziehungen. Unsere Beobachtungen zeigen, dass Code der auf eine besonderen Anforderung verweist, auch durch Aufrufbeziehungen verbunden ist. Wir nutzen diese Verbindungsmuster um Traces zu validieren. Die empirische Auswertung von vier Fallstudien von Systemen mit einer Größe von 150 KLOC und 59 Anforderungen zeigt, dass unser Ansatz die meisten Fehler mit $> 90\%$ Richtigkeit erkennt und die Qualität der validierten Traces nur sehr langsam mit fallende Eingabequalität nachlässt. Unser Ansatz ist voll automatisiert, Tool unterstützt und effizient (lineare Komplexität mit einer Laufzeit von < 3 Sekunden bei der Validierung der größten Fallstudie).

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Goals	3
1.3	Structure of this Thesis	4
2	Requirement-to-Code Traceability	6
2.1	Terminology	7
2.2	Usefulness of Traceability	10
2.3	Traceability Techniques	12
2.3.1	Static	13
2.3.1.1	Concern Graph	13
2.3.1.2	Abstract System Dependence Graph	15
2.3.2	Dynamic	17
2.3.3	Textual	18
2.3.3.1	Information Retrieval	19
2.3.3.2	Example IR Model: Vector Space Model	22
2.3.3.3	IR as a Traceability Approach	23
2.3.4	Hybrid	24
2.3.4.1	SNIAFL	25
2.3.4.2	CERBERUS	28
2.4	Trace Validation	32
3	Hypothesis of Surroundedness	36
3.1	Case Studies	37
3.2	Requirement Call Graph	39
3.3	Hypothesis	41
4	Observations	42
4.1	Clustering	42
4.2	Connectedness	44
4.3	Requirement Call Chain	45
4.4	Patterns	46
4.4.1	Surrounding Patterns	47
4.4.2	Boundary Patterns	49

5	Automated Trace Validation Approach	51
5.1	T over N Dominance	51
5.2	Boundary patterns vs. t/n-surrounding patterns	53
5.3	Algorithm	54
6	Tool Support: <i>TraZer</i>	58
6.1	Design Decisions	58
6.1.1	Eclipse as Platform	58
6.1.2	Database	60
6.2	User Interface	63
6.2.1	Perspective	63
6.2.2	Editors	65
6.2.3	Views	67
7	Evaluation	70
7.1	Accuracy	70
7.2	Coverage	73
7.3	Scalability	74
8	Discussion and Future Work	75
8.1	Incompleteness	75
8.2	Granularity	77
8.3	Traces Prediction	78
8.4	Conclusion	79
	Bibliography	80

List of Abbreviations

ASDG	Abstract System Dependence Graph
BRCG	Branch-Reserving Call Graph
CG	Call Graph
DFT	Dynamic Feature Traces
EF-ICF	Element Frequency-Inverse Concern Frequency
EPL	Eclipse Public License
GPL	GNU General Public License
IR	Information Retrieval
IDE	Integrated Development Environment
IDF	Inverse Document Frequency (<i>idf</i>)
LSI	Latent Semantic Indexing
MPL	Mozilla Public License
PDA	Prune Dependency Analysis
PDG	Procedure Dependence Graph
RCG	Requirement Call Graph
RTM	Requirement Trace Matrix
SEA	The Institute for Systems Engineering and Automation
SDG	System Dependence Graph
SPR	Scenario-Based Probabilistic Ranking
SR	Software Reconnaissance
TF	Term frequency (<i>tf</i>)
TPTP	Eclipse Test and Performance Tools Platform
VSR	Vector Space Retrieval
UI	User Interface

List of Figures

2.1	Domain overlapping between program comprehension, feature location, and requirement traceability	10
2.2	formal representation of a Multiplier class [24]	14
2.3	Concern Graph of the Multiplier class [24]	15
2.4	System Dependence Graph building process [6]	16
2.5	IR traceability recovery process [2]	21
2.6	A Sample BRCG: Source Code (left) Vs. BRCG (right) [29]	26
2.7	The Process of SNI AFL Approach [29]	27
2.8	Overview of the CERBERUS approach [10]	29
2.9	Performance of the 13 study participants plotted in the precision-recall space [18]	33
3.1	Caller and Callee relationships	39
3.2	Excerpt of Call Graph for Chess System	40
3.3	Requirement call graph for Chess System	41
4.1	Most methods implementing a given requirement are connected directly or indirectly by method calls	45
4.2	Call chain from a Chess RCG	46
4.3	Examples of “ $t \succ ? \succ t$ ” and “ $n \succ ? \succ n$ ” patterns.	47
4.4	A boundary pattern “ $t \succ ? \succ n$ ”	49
5.1	Combination of t-surrounding and n-surrounding on method Control.setPiece()	52
5.2	Errors inside and outside of requirements regions are detectable, but errors along boundaries of such regions are not.	53
5.3	Distribution of correctness over coverage.	56
6.1	DatabaseReference class diagram.	61
6.2	The <i>TraZer</i> perspective.	64
6.3	The RTM Editor.	67
7.1	Validation quality of traces/no-traces in poor quality RTMs.	71
7.2	Validation coverage for traces.	73

List of Tables

3.1	Properties of case study projects.	38
3.2	Excerpt of RTM from Chess system.	38
4.1	Likelihood of clustering	43
4.2	Percentage of connected trace/no-trace methods	44
4.3	Likelihood of calling relationship patterns	48
5.1	Seven categories of surrounding and boundary patterns	54
5.2	Coverage and correctness of categories in %	56

Danksagung

Ich möchte mich an dieser Stelle bei allen Personen, die mich während des Studiums und der Anfertigung der Diplomarbeit fachlich und mental unterstützt haben, ganz herzlich bedanken.

Besonderer Dank gilt meiner Familie, besonders meiner Eltern, die immer an mich geglaubt haben und mir während einer schwierigen Zeit zur Seite gestanden sind.

Ebenfalls gilt besonderer Dank Prof. Dr. Alexander Egyed für die großzügige Unterstützung und Betreuung bei dieser Arbeit und der damit verbundene Mühe. Ohne diese Unterstützung hätte diese Arbeit in dieser Form nicht durchgeführt werden können.

Achraf Ghabi

Chapter 1

Introduction

1.1 Motivation

Requirements-to-code traces reveal where in the code a requirement is implemented. This is important for many software engineering tasks including code comprehension and understanding the impact of requirement changes. Traces are said to be most useful in cases where developers are not/no longer familiar with the source code - a situation that tends to occur during maintenance where original developers may have left, or in larger software systems where code is manipulated by multiple developers. It has been shown in several experiments [25] that lack of understanding where a requirement is implemented leads to higher effort and more errors. This is not surprising because a developer who is not/no longer familiar with the source code is less capable of performing changes than a person familiar with the source code. It has also been shown that developers not familiar with source code are typically only able to locate about half of the code where a requirement is implemented [11, 18]. It is thus not surprising that software engineering suffers from this lack of knowledge, leading to changes at inappropriate places or unnecessary code replication. Keeping track of where requirements are implemented in the code is thus a fundamental best practice of the software engineering process.

Unfortunately, capturing and maintaining traces is a largely manual activity. Some automation exists but many approaches are not applicable to informal/unstructured requirements [13]. One example is information retrieval where requirements-to-code traces are derived through wording similarities between requirements and code [7, 8, 2, 16, 19]. However, the reported precision and recalls for these approaches vary widely and are typically dependent on the quality and quantity of descriptions that complement requirements and source code. Whether requirements-to-code traces are captured manually or through the help of such heuristic approaches, the quality of the

traces is unpredictable. Even if traces are captured by the original developers and are thus of high quality, these traces may deteriorate over time (with code and requirements changes) unless they are explicitly maintained.

This thesis introduces a novel approach for validating the correctness of requirements-to-code traces by exploring known trace knowledge in combination with patterns of calling relationships. Specifically, if a method has neighboring methods that are calling it (callers) or are being called by it (callees) and these neighboring methods are known to trace to a given requirement then the method in question is very likely to trace to that requirement also. We refer to this property as “surroundedness” (implying a method being surrounded by similarly tracing methods). As input, our approach obviously requires existing requirements-to-code traces, which need validation. Our tool implementation accepts such traces in form of a requirements-to-code trace matrix (RTM) where for each method and requirement it is defined whether the method traces or does not trace to the given requirement. Each cell in the RTM is thus a “trace link” that defines whether a method is implementing a given requirement (trace) or not (no trace). The RTM is expected to be complete but not expected to be (fully) correct. Furthermore, our approach requires knowledge which methods call one another (i.e., to be able to identify neighboring methods). Our tool implementation expects this in form of a call graph which identifies methods as nodes and calls as edges among these nodes.

For each method and requirement (each cell in the RTM) the principles of surroundedness are tested separately. If the principle applies then our approach computes an expected trace ('t') or expected no trace ('n') - depending on the patterns involved . If the principle does not apply for a given requirement and method then our approach fails ('fail') to compute an expected value. For all cells where the approach fails, our approach is not able to validate it (manual intervention is necessary). For the remaining cells, the approach simply compares the expected value with the actual value in the RTM and, if they differ, the user is informed of the error. The unique aspect of our work is that it considers method's calling relationships (which are presumed correct) together with the traces of surrounding methods (the caller and callee methods which are of unknown quality). Our approach is thus not an oracle that generates some traces automatically (e.g., like information retrieval [2, 15, 20] but instead searches for likely/unlikely combinations of traces and neighbors. Simply speaking, we found that certain combinations of traces together with calling relationships are not expected to occur often (a heuristic). Thus, neighboring method's traces need not be correct for our approach to function. Quite contrary, the fact that neighboring traces are included in the reasoning process makes it possible to identify unlikely situations which we report as errors .

Our approach was empirically evaluated on four case study systems – totaling over 150KLOC in size and covering different application domains. For these systems over 39 requirement-to-code trace links were available which we obtained from the original developers of these systems. In this work, we will present four patterns for identifying expected 'traces' and 'no traces'. Since these patterns have somewhat different likelihoods of correctness (the heuristical nature), our approach also reports the likelihoods of the detected errors being correct. Our empirical evaluation showed that our approach is able to validate about 93-96% of the 39 requirements-to-code traces and of these most of the requirements-to-code traces (about 77-89%) are validated with >90% correctness. In comparison, note that a random chance of guessing a correct expected 't' is just 5-12% because the requirements are often only implemented in small portions of the code and traces are thus rare in comparison to no-traces. To evaluate whether our approach is able to correctly validate traces of lesser quality, we additionally performed random seeding of errors and found that our approach performs well (near optimal) with the percentage of seeded errors exceeding 10, 20, 30% while the percentage of cells where our approach fails to compute an expected result increases - we refer to this as coverage. This implies that our approach is able to validate even poor quality RTMs well, however, with decreasing coverage. This appears an acceptable trade off since the quality is more important than quantity (poor quality implies more manual intervention anyhow!).

1.2 Goals

The goal of this work is to devise an algorithm for automatically validating requirements-to-code traces. As input, the algorithm takes a requirements-to-code trace matrix (RTM) where the matrix cells represent the individual relationships between methods and requirements. The contents of a cell is either a "t" (trace link) if the method implements the requirement or a "n" (no trace link) if it does not. In principle, a cell could also be empty (incompleteness) but for validating traces this is not useful and ignored. The algorithm for validating the traces should either confirm the contents of a cell (e.g., the algorithm agrees with a given 't' or 'n') or it should report an error (an incorrect 't' or 'n'). Detecting an error is synonymous with recommending a correction: if it determines an incorrect 't' then the algorithm expects a 'n' and vice versa. However, due to the informal nature of requirements and the complexity of the source code, it is more advisable that a human manually investigates error feedback. It is important to note that for trace validation, it is as important to validate a 'no trace' as it is to validate a 'trace' (an incorrect 'trace' is as much an error as an incorrect 'no trace'). Having more correct no traces would decrease the chances of having incorrect traces and vice versa.

During this work, we implemented a tool that supports the user in validating trace links. The user provides the input data (RTM and call graph) and the tool validate the cells in the RTM, reporting back false trace information. The interactivity with the user is very important. The tool must provide information about the expected value for a cell and hence let him double check the surroundedness property of methods. It is also important to measure the behavior of our approach on poor quality inputs, since validation is only required for RTMs known to contain possible erroneous trace/no trace links. Our approach must deliver reliable accuracy even when validating poor quality RTMs. We will verify the delivered traces expectations by testing multiple input RTMs with different correctness levels.

1.3 Structure of this Thesis

In *Chapter 2 - Requirement-to-Code Traceability*, we will enframe the area of requirement traceability that we cover in this thesis. First, we will define the specific traceability terminology, before explaining the usefulness of traceability in software engineering. Later, we will discuss the traceability topics that seemed to be related to our work.

Chapter 3 - Hypothesis of Surroundedness will be the first step into our work. We will explain the main problem of validating traces and our hypothesis on how are we intending to resolve it. We will also introduce case study systems that will be required to confirm or disprove our assumptions. This chapter refers also to fundamental techniques that are applied by other approaches presented in Chapter 2. These techniques are partly adopted to serve our approach.

Chapter 4 - Observations explains the different aspects that we were able to record on the case study systems using the techniques that we built in chapter 3 for this purpose. These aspects are the main motivation to our approach and helped us proposing an algorithm for validating traces.

Chapter 5 - Automated Trace Validation Approach presents our concrete trace validation algorithm and an empirical study of its performance on the case study systems. Of course, the results are different from one system to another, but the most important is that our algorithm delivers reliable correctness values.

Chapter 6 - Tool introduces the eclipse plug-in that we have implemented during this work. It is supposed to realize the automation of our algorithm and make the traces validation task more handy to users.

In Chapter 7 - Evaluation, we, obviously, will apply our approach on erroneous traces and evaluate its performance in different cases. We focus, thereby, on the aspects of accuracy, coverage, and scalability of our algorithm.

Chapter 8 - Future work discusses the open issues that we might investigate in future studies. The open issues might be limitless in this research area, but we will refer to those issues with which we also had to deal during this work.

Chapter 2

Requirement-to-Code Traceability

Although requirement traceability is mandated by many standards, many companies do not use it on their projects because of the costs following it. Requirement traceability requires an additional effort from developers during a project lifecycle. We might argue that the developer has just to explicitly set a link between the requirement and the code which he is implementing. This is basically what the developer has to do during the first implementation of a feature. We refer to this activity as traces retrieval. On the first sight, this operation might seem manageable with low effort. Unfortunately, the traceability tend to be more complicated during the life cycle of a software product. The continuous changes on source code might affect existing traces. In that case, we talk about maintenance of traces which is task that requires the most effort. Thus, software development companies have to deal with a trade of between the effort and the usefulness of requirement traceability.

Egyed et al. [11] showed that capturing traces is not a trivial task. They conducted an experiment in which developers having different experience levels tried to capture trace links between requirements and code in two example projects. The results of the experiment resulted in interesting insights. They showed that the effort required for traces increases with increasing complexity of the code. The participants in the experiments tended to make more mistakes when dealing with more complex code. The complexity is not referring to the asymptotic complexity of the code but rather the semantic facts, such as: the meaning of identifiers. Using inappropriate identifiers or unusual acronyms in the code would make it more difficult to understand, and consequently, the developer might be misled to define wrong traces or to oversee some of them.

Researchers are certain about the benefit that traceability would introduce to software development process once the effort is amortized. They developed many approaches with different levels of automation in order to minimize the effort required from a

human actor and let the computer retrieve the traces automatically. Among all the proposed techniques, some are also using the human-like approaches, such as: wording similarities between requirement definitions and source code identifiers [2]. Thus, using the appropriate terminology is not only important to humans but also to automatic techniques.

In this chapter, we will present a common introduction to traceability with the focus on the various available techniques for it. As we previously mentioned, there are already many techniques proposing how to capture requirement-to-code traces. We will discuss the most known techniques in order to show their advantages and also their limitations. This should help understanding how our approach for validating requirement-to-code traces would help overcoming many of these limitations, and hence it will improve the overall automation of traceability. But in order to understand these techniques the best, let us first explain the formal terminology related to requirement traceability.

2.1 Terminology

In the course of this thesis, we investigated traceability and we noticed that computer scientists are often using a specific terminology for requirement traceability. Sometimes, they were clearly referring to different things, but in other cases, they did apply synonyms for the same thing just because they are handling the same problem but from another perspective. In this section, we will explain the fundamental terminology about traceability that we continuously encounter throughout this work. Indeed, it is important to understand resemblances and differences between terms before talking about relating a technique to the one term or the other term.

From the first impression, a novice to requirement traceability would intuitively understand it as a research area dealing with links between requirement and source code, which are called *traces*. But in computer science, the term “trace” (noun) is usually used for execution traces and debugging statements. Formally, the IEEE Standard Glossary of Software Engineering Terminology ¹ has three definitions for this term:

- (1) A record of the execution of a computer program, showing the sequence of instructions executed, the names and values of variables, or both. Types include execution trace, retrospective trace, subroutine trace, symbolic trace, variable trace. (2) To produce a record as in (1). (3) To establish a relationship between two or more products of the development process; for*

¹<http://standards.ieee.org/findstds/standard/610.12-1990.html>

example, to establish the relationship between a given requirement and the design element that implements that requirement.

Requirement traceability refers to traces from the third definition (3). A *requirement-to-code trace* is a trace connecting a requirement to the source code implementing it or vice-versa. The trace relationship is commonly be-directional and serves to reach the source code from the requirement as well as the requirement from the source code. In this work, we derived another term, namely “*no-trace*”, which designates the opposite relationship of trace. When a no-trace relationship is set between a requirement and a source code, it means that the source code is not implementing that requirement (and vice-versa: the requirement is not implemented by the given piece of code). Many researcher used also other terms (e.g. relationship, link) to denote a trace. In this work, you might also alternate using these synonyms (relationship, link and trace) to denote a traceability relationship.

Requirement-to-code traces are usually gathered in a two-dimensional matrix where requirements are set on the one dimension and the code elements on the other. This kind of matrix is commonly known as *requirement trace matrix* (RTM). Although there is no standard format definition for RTMs, common sense practices recommended ordering the requirements in the columns and code elements in the rows. Actually, a two-dimensional matrix is nothing else than a table. Therefore, RTMs are often stored as tables in spread sheet editing tools which can be office tools, such as: Microsoft Excel², LibreOffice Calc³... The idea of putting the requirements on the columns side of a table could be related to the limitation of such tools (e.g. Microsoft Office 2007 supports only tables with less than 256 columns).

Requirement traceability is a software engineering research area covering activities that are related to traces, such as: capturing, recovering, and maintaining traces. The IEEE Standard Glossary of Software Engineering Terminology defines traceability as:

(1) The degree to which a relationship can be established between two or more products of the development process, especially products having a predecessor-successor or master-subordinate relationship to one another; for example, the degree to which the requirements and design of a given software component match. (2) The degree to which each element in a software development product establishes its reason for existing; for example, the degree to which each element in a bubble chart references the requirement that it satisfies.

²<http://office.microsoft.com/en-us/excel>

³<http://www.libreoffice.org/features/calc>

This definition denotes the general concept of traceability. Requirement traceability is a special application of that concept. The requirement engineering community have, therefore, their specific definitions. For example, Gotel and Finkelstein said that “*Requirements traceability refers to the ability to describe and follow the life of a requirement, in both a forwards and backwards direction*”[13]. The life of a requirement is obviously the transition from a simple requirement description in human language to an executable function in a program. In other words, requirement traceability deals with the information (or more precisely traces) connecting the requirements in their row state to their final executable state. This definition proposed by Gotel and Finkelstein covers also the scope of this thesis. Validating traces is an essential activity for capturing, recovering or maintaining such connections between different system models, namely requirement documentation and code implementation. For simplicity, we will refer to requirement traceability in the remaining of this work as traceability.

Traceability is a very active research area. It is excessively used for resolving many software engineering issues. Program comprehension and feature location are two research topics which are very related to traceability. People are often confused assigning new techniques to the one or the other research topic. We need to understand the boundaries of each topic in order to understand the usefulness of traceability. Biggerstaff defined program comprehension saying:

A person understands a program when they are able to explain the program, its structure, its behavior, its effects on its operational context, and its relationships to its application domain in terms that are qualitatively different from the tokens used to construct the source code of the program.[5]

In other words, program comprehension is the research area studying how to understand programs, their structure, their functionalities, and their applications. But understanding all these properties requires necessarily understanding the relationships between different components of a program. Thus, program comprehension needs the feature location techniques to establish the relationships between features (functional properties) and source code (structural properties) of a program (software system). But feature location is a completely separate research area that could also be used for other purposes, e.g. estimating the removal or addition of features.

We depicted in Figure 2.1 the overlapping between program comprehension, feature location, and requirement traceability. Feature location is covering a big part in program comprehension and almost completely incorporating requirement traceability. We discussed the need to feature location in program comprehension, but how could we explain the relationship between feature location and requirement traceability? In order to answer this question, we need to understand the difference between a feature

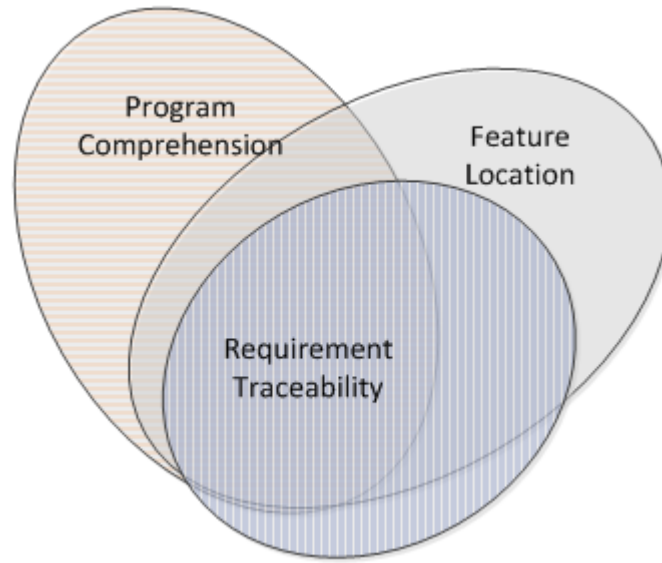


Figure 2.1: Domain overlapping between program comprehension, feature location, and requirement traceability

and a requirement. The IEEE Standard Glossary of Software Engineering Terminology defines a software feature as:

... A software characteristic specified or implied by requirements documentation (for example, functionality, performance, attributes, or design constraints).

A feature is, thus, a set of system requirements, and hence the difference between a feature and requirement is only a question of granularity. A requirement is very granular and directly related to the low level (implementation) functions, but a feature is rather a general definition of a high level functionality which is usually realized by multiple low level functions. Features are often called concerns because they are of higher significance for external stockholders of a software system (e.g. client, user...).

2.2 Usefulness of Traceability

In order to talk about usefulness of traceability, we have to assume that traces are available, correct, and complete. Under that same assumption, Winkler and Pilgrim [27] summarized important application scenarios of traces from various scientific reports and research contributions. They were basically able to identify 16 use cases. In

the following, we list from them the ones that we consider as general for software products:

Estimating change impact Traces provide direct links to change locations when proposing requirements changes. Knowing where a requirement is implemented, the developer will be able to better estimate the effort needed for a certain change.

Proving system adequateness Through traces, the customer should be able to identify the range of implemented features. This makes him sure that he is getting the requested functionalities and he is not paying for implementing a code to which he is not in need.

Validating artifacts Multiple defects could be identified by applying traces on artifacts level. A simple traces check could ensure whether an artifact is implementing the intended specification or not. Defects, such as: inconsistency and incompleteness, should be detectable at a very early stage of an artifact life cycle.

Testing the system Because, traces provide a direct reference between the tested code and the requirement it is implementing, the developer can easily consult the appropriate specification in order to create suitable tests cases and meaningful test data.

Supporting special audits or reviews This is the main reason for including traceability in the *NASA Software Assurance Standard*. The traces are mandatory to perform audits or reviews of critical parts in a system. Without traces, this task could be practically impossible on huge software systems.

Improving changeability Having an overview of artifacts interconnections with their corresponding requirements should help when designing new changes on the system, and especially when the changes are crosscutting multiple requirements and/or artifacts at the same time.

Monitoring progress The amount of traced code and requirements could be used by the management as a metric about the overall progress of a project. The manager can identify missing implementations and tests for untraced requirements.

Understanding the system For stakeholders with limited information about a system, the traces could be practical instruments to understand the system from different perspectives. For example, the architecture of the system would be very clear to an external developer when she sees the link between the requirement documentation and the code implementing it.

These are only example of traceability use cases and there are many others. As mentioned before, there is a tradeoff between effort and usefulness. A company should identify the benefit that they are expecting from performing traceability before even starting to capture traces. The estimated cost for such a task has been always the most cruel rival of using it, even though Egyed et al. [11] have shown that its overall effort is very limited. Many researches were set to investigate this problem and propose techniques that at a time, reduce the effort and improve the quality of traceability.

For a program, call graph could be generated in multiple ways. On the one side, there are static call graphs that are generated out of the structural properties of a program, such as: concern graph (see section), or abstract system dependency graph (see section)... On the other side, there are dynamic call graphs that are generated at runtime depicting executed call relationships between code elements. In this thesis, we decided to use dynamic call graphs due to multiple reasons. Compared to static graphs, a dynamic call graph would contain only needed code and thus, it truncates implicitly dead code and reduces the size of the graph which would improve the performance of any algorithm running over the graph structure.

2.3 Traceability Techniques

In last two decades [9], there has been a lot of research activity about traceability issues, especially requirement-to-code traceability. Multiple techniques has been presented where different approaches are applied to retrieve and/or maintain trace links between code and other software artifacts. Revelle et al. [22] classified state of the art feature location techniques into three main core types: static, dynamic, and textual analysis. These types do also apply for traceability techniques. Textual analysis has been mainly exploited by performing Information Retrieval techniques on textual software artifacts (e.g. source code, developer comments, and documentation) [2, 15, 20]. A Static analysis method was proposed by Chen and Rajlich [6]. The method should help a user navigating on a static abstract system dependency graph which would help her understand and retrieve implementation locations of different requirements. Other approaches [17, 21, 24] investigated the dynamic information acquired from programs. Profiling tools are used to gather execution information of software artifacts. Dynamic techniques analyzing this kind of information with the aim to provide a better understanding of the requirements implementation in a program. There are also hybrid methods that combine multiple techniques into one approach. Eaddy et al. [10] presented a very good example of combining the three technique types (static, dynamic, and textual) in order to improve the quality of trace recovery process in their tool (CERBERUS). Calling relationships in particular have also been used to improve

the reported ranking of traces in information retrieval approaches [8, 2, 22, 10] where higher-ranked traces are presumed to be more correct traces.

In the following sections, we will present few approaches implementing the different types. At this stage, it is important to understand the effort and quality related to each approach.

2.3.1 Static

Multiple conventional techniques (grep, IDE, database...) could statically identify relationships between different elements in source code. Developers are usually applying these static techniques for locating and understanding features in a program. The tools supporting one or multiple of these techniques provide always the same kind of information. They deliver references to the lines in source code that represent a perfect match to the feature in question. E.g. when a developer searches for the term "Person" in the source code of a program using "grep" function, he would get all line of codes where this term occurs. Some sophisticated IDEs (e.g. Eclipse) might allow the developer to provide additional meta-information about the searched item. E.g. he might define that the term should be searched only among class names and thus, the tool would deliver only classes having the term "Person" as part of their names. The assistance provided by these techniques is very "source code-intensive" and could not present an infrastructure for reasoning about or analyzing different features and their relationships at the same time. Therefore, multiple researches were concerned with creating a more helpful static presentation of features. Robillard and Murphy [24] developed a Concern Graph as an abstract model for programs that should help documenting and analyzing features (concerns). Few years later, Chen and Rajlich [6] proposed another graphical representation of source code called *Abstract System Dependence Graph* (ASDG) aiming at helping developers during the feature location task.

2.3.1.1 Concern Graph

Robillard and Murphy [24] proposed the Concern Graph approach, a structure-oriented feature representation. It is intended to provide a better "feature description based on the relevant program elements and their relationships" [24]. The Concern Graph presents an abstract code model by omitting implementation details. It focuses mainly on dependencies between code elements which makes the overall implementation of a feature more understandable to a developer.

Object-oriented paradigm defines a very clear program's structure. Even with most complex function implementations, the code remains relatively easy-to-overlook in a way that code elements (classes, methods, and fields) and the interaction between them can be identified and classified with minimal effort. The concern graph concept uses this basic idea to provide a formal model of programs. A concern graph is defined as $P = (V_p, E_p)$, where V_p is the set of vertices, and E_p is the set of labeled directed edges $e = (l, v_1, v_2)$ [24]. A vertex is representing exactly one of three code types: Class Vertex "C" (global class without considering its fields or methods), Field Vertex "F" (field of a class), or Method Vertex "M" (method of a class). Edges in P are reflecting calling relationships in the source code by connecting vertices of calling elements to their respective called elements' vertices. We identify six types of edges: (M, M) , (M, F) , (M, C) , (C, C) , (C, M) , and (C, F) . Figure 2.2 shows a concrete representation of a simple Multiplier class. The edges are additionally labeled with tags referring to the semantic relationships between elements. E.g. a "calls" tag from method `product()` to method `sum()` means that the method `product()` is directly calling the method `sum()`. In the same way, a model representation of the complete program would be created.

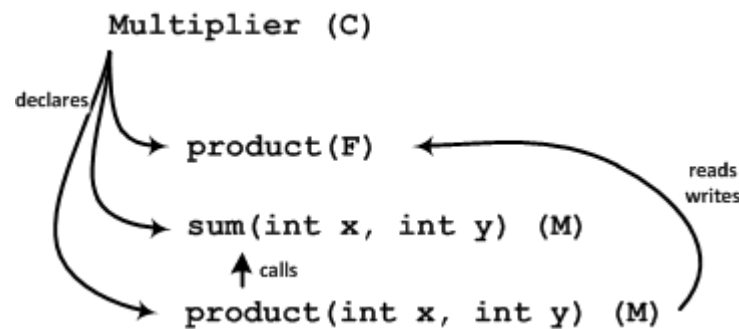


Figure 2.2: formal representation of a Multiplier class [24]

A Concern Graph would include only vertices of elements related to the implementation of a specific concern and the edges connecting them. Usually, commercial software products have multiple interconnected concerns. The graph presentation P of a program should incorporate all its implemented concerns (features) in order to provide an appropriate presentation of the entire program. Each concern will be presented by its own Concern Graph similarly to the Multiplier example (see Figure 2.2) which is a compact subset of the graph represented the entire program. In the complete program graph, there are two types of vertices. One type encloses vertices which are shared between multiple compact graphs. They express shared code elements between concerns. These vertices designate usually the *part-of* relationship among concerns. In case of removing a feature, we would have to keep the shared code elements because they will be needed by other concerns. The second type encloses the vertices which are specific for a single feature, and hence they are only available in its concern graph. These are

known as *all-of* vertices. They do only exist for that feature and if we would remove the feature, we could also remove them without influencing other concerns. Robillard and Murphy [24] define a Concern Graph of a feature "i" formally as $C_{p,i} = (V_{p,i}, V_{p,i}^*, E_{p,i})$ of a Program $P = (V_p, E_p)$, where $V_{p,i}$ and $V_{p,i}^*$ are respectively distinguishing between part-of and all-of vertices. $E_{p,i}$ designates the edges subset from (E_p) connecting the $V_{p,i}$ vertices. For simplicity, the details that could be unambiguously derived from P are omitted. E.g. An all-of class vertex and the edges connecting it could be automatically derived from its methods vertices and thus, there is no need to add that class vertex to $V_{p,i}^*$ or its edges to $E_{p,i}$. Omitting this kind of information should keep the concern graph relatively compact and manageable.

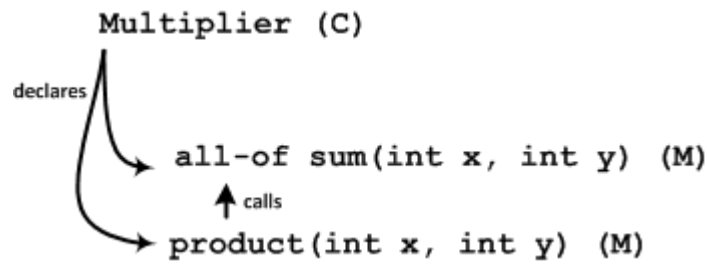


Figure 2.3: Concern Graph of the Multiplier class [24]

Considering our example from Figure 2.2 as a formal representation of feature for multiplying numbers, we would generate a Concern Graph as in Figure 2.3. The graph shows addition meta information (*all-of*) that should help the developer understand the importance of code elements for the actual concern. Information that seems to be obvious (e.g. `Multiplier` declares the field `product`) or that could be derived automatically (e.g. method `product(int x, int y)` reads-writes field `product`) are omitted in order to keep the graph as compact as possible. The developer could generate these graphs automatically and expand them later with additional meta-data details. He should consider the tradeoff between compactness and expressiveness of the concern graph. Striving for simplicity should not deteriorate the understandability of the graph for other user.

2.3.1.2 Abstract System Dependence Graph

In the early 90's, Ottenstein [28] proposed the *procedure dependence graph* (PDG) for non-object-oriented programming languages, such as: C ... It introduces a graphical presentation for single procedures. The vertices represented code elements which are either single statements or sets of statements. The edges, on the other side, represented to dataflow between the different code elements in the vertices. Few years later, the

system dependence graph (SDG) [1, 14] was proposed by combining several PDGs. Additional edges were introduced in order to model function calls and parameter passing between procedures.

Unfortunately, SDG is very granular and contains more details than needed for features location purposes. Chen and Rajlich [6] proposed to reduce the details level and make it more abstract. They introduced the *abstract system dependence graph* (ASDG) which is a less granular subset of the original SDG. Vertices symbolize as usual the code components (sets of statements selected by a developer). They also distinguish between two types of edges: *call* edges that designate a conventional procedure call; and *data flow* edges that designate a global variable access (read or write). Although, both SDG and ASDG are very similar and could be build in the same way (using the same algorithm), the automated support is be very limited. The developer have a central role taking decisions about the graphs' structure.

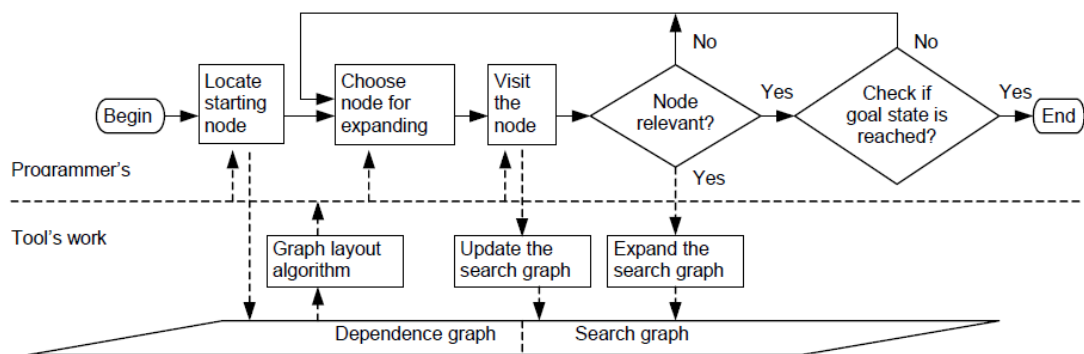


Figure 2.4: System Dependence Graph building process [6]

Building an ASDG begins by selecting a starting code element (component) in the source code. Usually, this would be the starting function `main()` of the program. Then, the developers selects a next component to visit in order to expand the graph. At each built vertex, all other code element (including global variables), that are reached by the current code element, must be directly connected. The programmer has to be sure that all neighboring code elements have been found and connected. A parallel temporary *Search Graph* could be built in order to keep track of components that are already selected and connected. This process will be done recursively until all components are selected (goal state). A tool might assist the developer during this task allowing him to undo and redo some of his decisions. Figure 2.4 depicts a more detailed process for building ASDG from source code. It also distinguishes between the tasks that should be done by a programmer from those that could be performed automatically by a tool.

2.3.2 Dynamic

Instead of static code analysis as in static techniques, the dynamic traceability techniques are analyzing programs at runtime. The information acquired from programs behavior at runtime should also help identifying the code responsible for given requirements. Software products are usually tested by running automated or manual unit tests. The automation level of tests depends on the kind of function being tested. Background functionalities (e.g. testing a calculation for correctness) are easy to automate because they do not need any user interaction. Manual functionalities (e.g. user interface testing) require user intervention and hence they are hard to be fully automated. Commonly, a program will be executed in order to verify the operability of its functions. Each requirement would be tested by automatically or manually calling the functions implementing it. Multiple unit tests could be invoked in order to cover a single system requirement. But in all cases, each unit test runs only a specific part of the system that we call an *execution slice* [28]. The piece of code implementing a certain requirement could be intuitively identified by comparing execution slices resulting from unit tests related to that requirement. The execution slice delivers a set of code elements that have been executed during the test. Comparing that set of elements with elements acquired from other requirements testing should identify the elements that exclusively needed by that requirement.

Based on this simple idea of execution slices, Wong et al. [28] proposed an algorithm for locating requirements implementation in the code. The algorithm consists of three steps:

1. Find the set of tests that invoke the components implementing a feature F (including tests that are shared with other features).
2. Find the set of tests that exclude components implementing F . Some tests could just exclude few components and not all of them.
3. Subtract the excluding set from the including set in order identify the elements that are uniquely related to F .

The first and second steps are the most difficult. The software analyst should be aware of many details about the requirements in order to be able to select unit tests that include or exclude a certain feature. The features themselves could be described in way making them hard to be separated from each other. Some features could be composed of smaller “sub-features”. When we subtract sets of executed code elements, we should consider the coverage of the selected tests over the requirement in question. Especially, the exclusion tests have to be “absolutely” excluding a requirement and not only few

aspects of it. Otherwise, the subtracted elements would be not only omitting foreign code but also code elements implementing the requirement under study. Therefore, the first and second steps of the algorithm must be carefully performed. Poorly selected unit tests might lead to inaccurate or even wrong results.

Whether the unit tests are fully automatic or manual, the execution information must be recorded when running them. Depending on the programming language used, there are different technologies that could be applied for this purpose. e.g. Java has a very sophisticated debugging interface which allows to record methods call information without changing the original source code. Wong et al. [28] used χVue for the C programming language. χVue instruments the code and builds an executable version of the program with logging statements that should gather information about code elements at runtime. The thirds step of a algorithm will use the information about each test from the execution log. The set of code elements called at runtime for a certain requirement will be subtracted from code elements that were also called by other requirements' tests.

Unit testing is a common practice in software development. The main effort needed for the dynamic feature location technique would not reside in creating unit tests but rather the careful selection of the appropriate ones in order to make ideal execution slices for subtractions. In some cases, this operation makes the developer aware of missing test scenarios which she has to retroactively implement them. The overall effort needed is still acceptable and beneficial, which is a big advantage counting for this technique compared to static approaches presented in previous section . Furthermore, the dynamic approach could be fully automated depending on the kind of features being analyzed. Many functional features are usually called internally without requiring a human interaction. Furthermore, recording the execution information for such feature would not require manual intervention as the case in static techniques. The only drawback that we could identify is that the accuracy of the results is directly proportional to the quality of selected tests. We might miss code elements implementing a certain feature just because we did not select the appropriate tests for it. In other words, we risk having problems with the completeness of the unit tests and the coverage they provide to the tested features.

2.3.3 Textual

Both static and dynamic techniques mandated specific knowledge about the system in question. Although, they reduced the overall effort needed for traceability, new burden manifested when realizing these technique for industrial use. There are major impediments for a general acceptance of these practices. None of the techniques presented

in previous sections was able to guarantee a certain level of accuracy. Multiple recent researches have focused on creating metrics to measure the success of their techniques relatively to others. At the same time, increasing the automation of both: the creation and maintenance of traces, remained very important goals of every new technique. As text mining and *information retrieval* (IR) showed a very good success in automatically extracting information from textual data, it was a very obvious step to try them also apply them for traces recovery. The source code and requirements documentation are -after all- textual data. We found multiple traceability approaches based on Information Retrieval (IR). Antoniol et al. [2] presented a method to recover traceability links between source code and text documents based on a previous study [1] about the different IR probabilistic models. Huffman Hayes et al. [14] proposed a method for improving candidate traces generation using IR. In this section, we will cover the general concept of IR approaches and present a sample model that has been used in several techniques.

2.3.3.1 Information Retrieval

Information Retrieval techniques are concerned with providing systems for retrieving documents relevant to a given query from a large documents database. The documents are commonly indexed and classified before storing them into the database. The common indexing process is as follows:

1. Each document is transformed into a character stream by removing non-textual (e.g. punctuation, pictures...) content.
2. The resulting text will be *lexically analyzed* in order to separate the different tokens (terms) from each other.
3. *Stopwords* (e.g. the, and, or...) are eliminated. The list of stopwords is language specific and could be shortened or extended depending on the requirements of the system.
4. Terms are normalized using a *Morphological Analysis* (also called stemming); e.g. context dependent terms, such as: “retrieve” and “retrieval”, are unified to a context independent root term “retriv”.
5. *Thesaurus* is applied for identifying synonyms and homonyms.

6. Each term will be weighted by calculating a numeric weight value usually based on the occurrence and/or frequency of that term in the given document or all documents in the database.

In a IR system, the user formulates a query in human language and expects the system to deliver the documents meeting his request. Matching documents to a query are said “relevant”. IR systems have different probabilistic models to calculate the relevance of documents in the database to the given query. The relevance of each document is computed from the likelihood of terms occurring in the query and the document. The relevant documents are selected from the database and ordered based on their probable relevance. The simplest model is to select documents containing the same terms as in the query and then to order them depending on the weights and number of occurrences in each document. But there are other more sophisticated probabilistic models applied for retrieving and ranking the documents, such as: Latent Semantic Indexing (LSI) and Vector Space Retrieval (VSR).

Terms weighting has a very important role in IR. It is the main parameter used for the retrieval as well as the ranking of documents. The ranking is intended to put the most likely documents in the top of the list of retrieved documents. This is intended to let the user find the information that he is looking for more quickly. Term frequency (tf) and inverse document frequency (idf) are well known terms weighting metrics in IR. The term frequency $tf_{i,j}$ represents simply the number of times a term j is appearing in a document D_i . The inverse documents frequency idf_j designates the inverse ratio of documents containing the term j over the entire set of documents. The concept of using idf_j is that the more documents where a term appears, the less important it is, and vice-versa, the less document containing that term the more important it is. Formally, idf_j is defined as:

$$idf_j = \frac{\text{TotalNumberofDocuments}}{\text{NumberofDocumentscontainingthe}j^{\text{th}}\text{term}} \quad (2.1)$$

The quality of an IR approach is measured by the relevance of the documents it returns when a query is called. The relevance is commonly expressed by two metrics, namely Precision and Recall. Recall is the ratio of the retrieved relevant documents for a given query over the total number of relevant elements for that query (see equation (2.2) on page 21). Precision is the ratio of the retrieved relevant elements over the total number of retrieved elements (see equation (2.3) on page 21)). Both metrics are commonly mapped to values between 0 and 1 which can be displayed also as a percentages. The higher the value, the better quality the metric does designate. i.e. A higher recall value for a query means that a higher number of relevant elements for that query are retrieved; A higher precision for a query result means that a higher number of the retrieved elements are relevant for that query.

$$Recall(Query_i) = \frac{\sum_i \#(Relevant \cap Retrieved)}{\sum_i \#(Relevant)} \quad (2.2)$$

$$Precision(Query_i) = \frac{\sum_i \#(Relevant \cap Retrieved)}{\sum_i \#(Retrieved)} \quad (2.3)$$

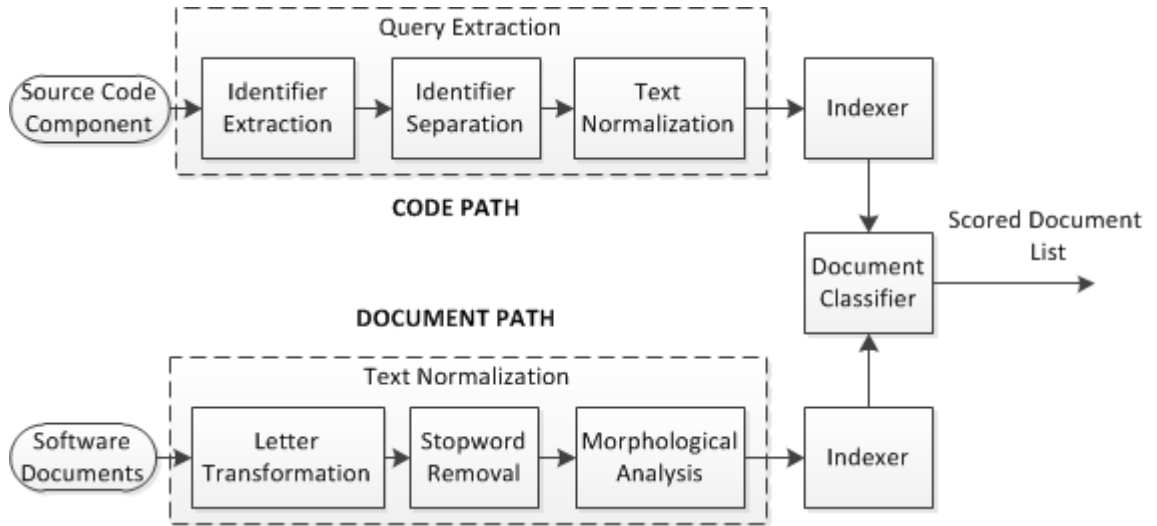


Figure 2.5: IR traceability recovery process [2]

We believe that IR techniques can help recovering traceability links between the source code of a system and its documentation. However, this could not be possible if the code does not contain meaningful terms that could match the documentation’s terminology. Indeed, the code and documentation of application share the same domain knowledge and consequently, they should deploy the same, or similar terminology. Antoniol et al. [2] presented a method that uses classes identifiers (e.g. class name, field names, method names...) as query terms to retrieve the documents relevant to a particular class. They employed a vector space model to weight terms in the documents and rank the documents later when they are retrieved. The overall process of traceability recovery is depicted in Figure 2.5. The figure consists of two paths: the upper one does extract the query from the source code, and the lower one is about preparing the documents for retrieval. Both paths end at a common classifier which is responsible for calculating the similarity between queries and documents. The classifier creates the final ranked list of relevant documents.

The lower path in Figure 2.5 is a common process for indexing text documents as defined by conventional IR techniques. The upper is, however, more specific for the traceability approach. The final part about indexing the documents and the queries as well as ranking the results would be implementing a particular IR model. Antoniol et

al. [2, 1, 3] applied different probabilistic models using this same architecture. They analyzed the effect of each model by adapting the classifier and indexers from at the end of the process. In this work, we refer to the vector space model presented in the paper [20] as an example application.

2.3.3.2 Example IR Model: Vector Space Model

In a vector space model, each document and query are mapped to a vector in an n -dimensional space, where n is the number of terms in the vocabulary extracted from all documents. Each vector is represented by an n -tuple corresponding to the weights of terms in the vocabulary. E.g. if we have the vector $[d_{i,1}, d_{i,2}, \dots, d_{i,n}]$ representing a document D_i , and $d_{i,1}$ corresponds to the weight of the first term in the vocabulary. The weight of terms could be computed in different ways. The simplest case would be using Boolean values where 1 means that the term occurs in D_i and 0 otherwise. Antoniol et al. [1] applied a vector model specific weight value which is derived from $tf_{i,j}$ and idf_j (see previous section). For each document D_i a weight $d_{i,j}$ is calculated for each term j as follows:

$$d_{i,j} = tf_{i,j} \times \log(idf_j) \quad (2.4)$$

There is a basic concept in IR that says: “The more the word is specific to a document, the higher the weight it should get”. In order to implement this concept, the $\log(idf_j)$ is acting as a normalization factor for the term frequency $tf_{i,j}$ in the document D_i over the entire set of documents in the database.

The query is similarly represented by a vector $[q_1, q_2, \dots, q_n]$ mapping its terms to the vocabulary retrieved previously from all documents. But, the terms that are found in the query and not in the vocabulary are simply ignored. In the case of Antoniol et al. [1], each query Q is composed of identifiers extracted from a class. The relevance of a document D_i to Q is computed by a similarity function that calculates the cosine of the angle between their corresponding vectors in the n -dimensional space:

$$Similarity(D_i, Q) = s_{i,Q} = \frac{\sum_{j=1}^n d_{i,j} q_j}{\sqrt{\sum_{h=1}^n (d_{i,h})^2 \times \sum_{k=1}^n (q_k)^2}} \quad (2.5)$$

The similarity value is also used as a relevance metric. When we call a query on the database, the similarity value will be calculated for each and every document in the database to that query. The higher the similarity value, the more relevant the document could be. Therefore, the documents will be ranked in a list by their relevance to the query. The most relevant ones must be on the top of the list. This strategy is, however, useless for a software analyst. On the one side, the complete list of documents (requirement

descriptions) would be annoying for the user even when they are ordered. On the other side, the operation of returning all documents is very resource intensive and could be inefficient. Antonio et al. [1] used a threshold parameter which is a common practice in IR techniques. The threshold defines where to prune the ranked list for a given query. The documents with a similarity value less than the threshold are ignored. The threshold is usually defined as a percentage relative to the best similarity. E.g. if we have a query result with a maximum similarity value of 0.60 and a threshold of 30%, then we return all documents with a similarity value greater or equal 0.42 ($= 0.60 - (0.60 \times 30\%)$). The threshold sets a tradeoff between precision and recall of the returned results.

The vector space approach was tested [1] on two example projects (LEDA and Albergate) and proved that it was able to retrieve most of the relevant documents (100% recall) at a certain threshold. The precision was, however, decreasing with the increasing recall. E.g. for LEDA⁴ library (Library of Efficient Data Types and Algorithms) project 100% recall was reached with 3.92% Precision: 2496 documents were retrieved but only 98 from them were really relevant to the given query. In this case, the user has to manually validate 2398 wrong traces in order to get all correct ones.

2.3.3.3 IR as a Traceability Approach

Information retrieval has many advantages compared to other techniques. The main advantage is that IR requires only textual input without any additional knowledge. Indeed, there are a lot of textual artifacts in software products (e.g. code, documentation...) that might support such techniques. The IR approaches are very useful, especially, when there are no former developers to help analyzing and tracing a program. E.g. many systems are often no longer supported by the original companies and the clients find themselves hiring new software analysts and developers in order to fix a problem or extend existing features. IR represents best possible help in such cases. It generate candidate links between code and documentation that should help developers finding the implementation of certain functionalities. The developer would be able to at least identify an entry point to understand the structure of the system, even though the retrieved links are not all correct.

Information retrieval is obviously reducing the size of the information used to be manually processed when retrieving traces between code and documentation (requirements). But despite the high automation of this technique, a software analyst is still needed to validate the proposed candidate links. Only human cognitive capabilities is able to confirm or reject a requirement-to-code trace. The results returned by an IR approach

⁴<http://www.cs.sunysb.edu/~algorithm/implement/LEDA/implement.shtml>

might reach very good precision and recall. It is, nonetheless, impossible to automatically eliminate all erroneous links. Considering the advantages of IR, many researchers took the challenge of using it to improve traceability approaches. E.g. Hayes et al. [14] proposed an approach that enhances the IR process at two levels: on the one side, they introduced a thesaurus in order to unify synonyms and improve the query semantic; on the other side, they considered the interdependency between queries in order to build relationships between queries. They assumed that the queries should have internal relationships similarly to code elements. The results proved to be very promising. Unfortunately, we cannot compare between the improved approach of Hayes et al [14] and the previous approach proposed by Antoniol et al. [1] because each research team included a different data set for testing. A comparison based on different data sets would be unfair.

The application of IR has also multiple limitations. First of all, IR techniques requires a huge amount of textual data in order to deliver reliable results. It expects to be fed with a big number of documents when using it for conventional text search tasks. In software projects, getting such an amount of textual data is possible only with medium and big projects. This technique could be completely unreliable for small projects. Furthermore, source code and requirement documentation must share meaningful names in order to find matching links between them. But, the naming convention is not always trivial to developers. Many developers work on the implementation of software products without even reading the requirements documentation. Usually, developers get their tasks from a manager who is aware of all details related to the requirements. The developers do not need to read the documentation individually. Thus, the terminology used in the source code would be directly depending on the instructions that the developers get from their manager. An implicit “common sense” of selecting identifiers when implementing a software product is often not enough to get an optimal textual similarity between the documentation and the source code. Nowadays, many big companies are aware of the importance of naming conventions and they define very strict naming conventions for such purposes.

2.3.4 Hybrid

The research in traceability area exploited many techniques from different types: static, dynamic, and textual. Although, each technique proved to be useful in some ways to capture or recover requirement-to-code traces, their accuracy is still to not convincing enough and have to be improved. Researchers decided to make hybrid techniques by combining several existing ones. They exploited the benefit from some combinations and how they would improve the quality of results compared to individual techniques separately.

In this section, we will present two hybrid approaches using different combinations, namely SNIAFL [29] and CERBERUS [10].

2.3.4.1 SNIAFL

The SNIAFL approach was proposed by Zhao et al. [29] in 2004. It is a software analysis process aiming to locate feature in the source code by combining the information retrieval (textual technique) with a static analysis technique, called Branch-Reserving Call Graph (BRCG). BRCG was initially developed to detect use cases of a program from source code [23]. The BRCG was intended to enhance system comprehension rather than locating features in the code.

Zhao et al. [29] define the BRCG as an extended form of a call graph. In addition to call relationships between functions (methods), it includes further information about branch statements. A branch statement is a code statement that creates a branch in the logical structure of the program. For example, an if-statement in java starts explicitly a new branch in the flow of a program with a curly bracket “{” and closes it by closing the bracket “}” again. Some programming languages (e.g. Java) allow defining implicit branches (without delimiting braces). Analogically, other statements, such as: case-statements, loop-statements...define also their own branches.

Figure 2.6 shows a BRCG for a sample java method `foo()`. The nodes in the graph are representing either a function (method node), a branch statement (Bi node), or a return statement (RS node). The relationships between nodes are classified whether as branching relationship, which designates the begin of a sub-branch, or a sequential relationship, which designates the sequential call of functions. They have always a relationship that connects them to each other. They cannot be directly connected to each other. Branches that are created by loop statements (e.g. `while`, `for...`) are handled as sequential relationships. The BRCG of an entire system is built in a similar way. The root node of a BRCG representing the entire system would be the entry method of the program (e.g. `main()` method in java programs). Usually, graphs of single methods are created first, and then they will be connected to each other in order to create an entire system representation.

SNIAFL proposes a process with four main steps in order to recover traces from IR information combined with the information gained from BRCG (see Figure 2.7):

1. **Acquire Initial Specific Connections between Features and Functions**

In this step, IR is applied to recover initial candidate traces between features and functions (methods). The approach uses the vector space model, which we have

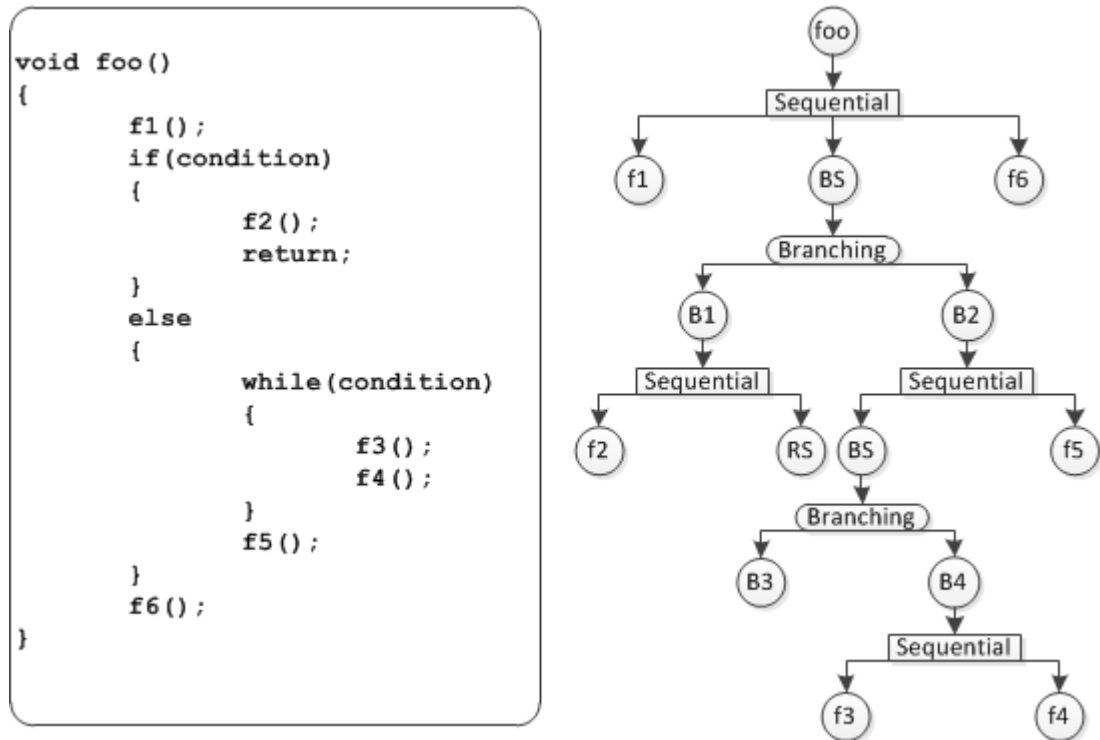


Figure 2.6: A Sample BRCG: Source Code (left) Vs. BRCG (right) [29]

already explained a previous section, for indexing and ranking documents. This approach relies, however, on filtered data. On the one side, the document set does not constitute of documentation pages, but rather of selected paragraphs describing single features. The concept of document in IR techniques is equivalent to a feature description in this case. Each feature will be represented by a paragraph written in human language. The description could be acquired from the requirement documentation of the system or from an expert who has enough knowledge about the requirement. On the other side, the query set is prepared from methods (functions) instead of classes. The set of identifiers (including the name of the method) appearing in the body of a method are extracted automatically and grouped into a function description that will be used as query. After preparing all documents and queries, initial connections (initial candidate traces) are retrieved by running each query on the documents set. This operation will select and rank the set of documents (feature descriptions) by their relevance to each query (function/method). The result list could also be empty if the similarity function does not return positive values for any feature description in the documents set.

2. Identify Initial Specific Functions

After identifying candidate feature descriptions for each function, we associate functions with their candidate features. In this way, we inverse the relationship

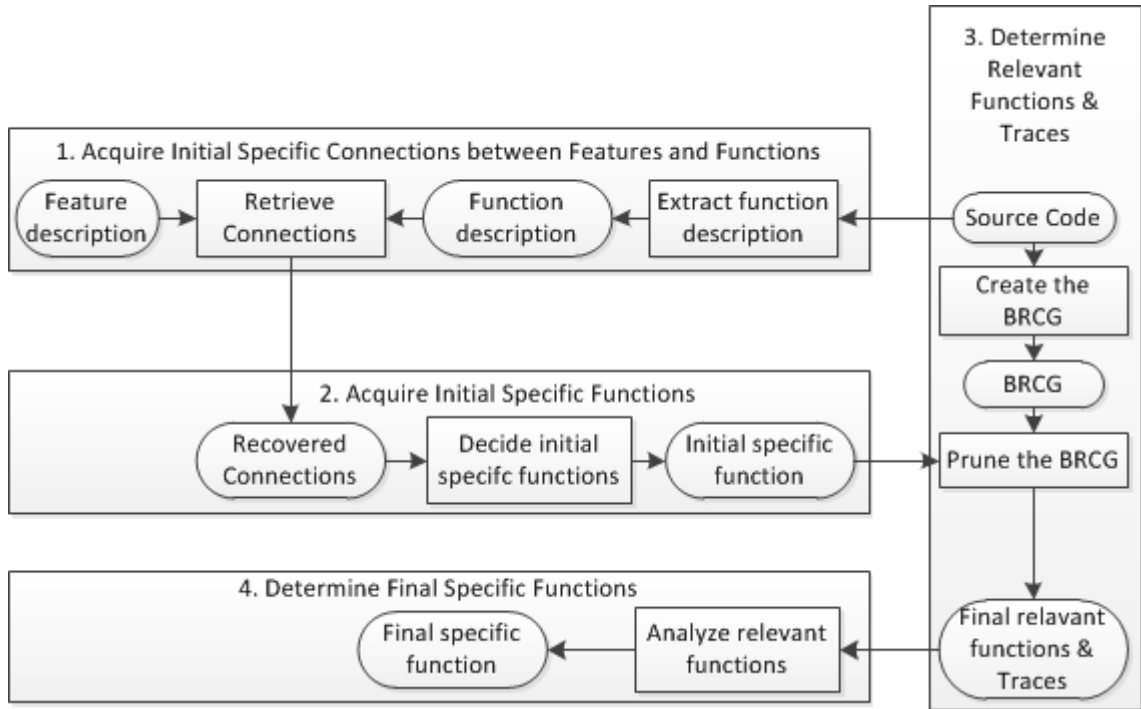


Figure 2.7: The Process of SNI AFL Approach [29]

between them and create new lists of candidate functions implementing each feature instead of candidate features describing each function. The similarity value calculated by the vector space model in previous step is also used to sort the functions by their relevance to the feature descriptions in a descendent order. Furthermore, a *rank distance* will be calculated between each two successive functions. The position where the biggest rank distance occurs in each list is considered as the best point to prune that functions list. The resulting lists are shorter and contain only most relevant functions. These are *initial specific functions* that are more likely to have trace connections to input features.

3. Determine Relevant Functions & Traces

In this step, a BRCG is created from source code. It will be reduced to the specific functions that have been identified as relevant in previous step. Every branch in the BRCG containing irrelevant functions is pruned. The resulting graph should contain only functions relevant to features. After that, both functions and features are summarized in a *requirement trace matrix* (RTM) where the rows denote the features and the columns denote the functions. Each function that is relevant to a single feature will be assumed to be tracing to it and a trace tag (in this case 1) will be inserted in the corresponding cell in the array. All other functions, including those that are shared between multiple features, are tagged in the array

as not tracing (a 0 is set in the corresponding cells). The *final functions and traces* are designated by the cells with the trace tag “1”.

4. Determine Final Specific Functions

Pseudo execution traces are generated from the reduced BRCG by traversing each possible path in the graph starting from the root. Every function with an assigned trace to a feature will propagate its traces to the following functions on the same execution path. Thus, the final RTM will be filled automatically with addition candidate traces. The effort required for this operation is directly proportional to the size of the input BRCG. Handling a big BRCG might be a resource intensive task.

The approach was tested [29] on an example system that also has been analyzed with bare information retrieval technique and dynamic technique separately. The IR results were very poor and did never exceed 50% for either precision or recall. The dynamic approach was inspected with different types of unit tests: insufficient tests and well-designed tests. In either cases, the dynamic analysis performed better than IR, especially with the well-designed unit tests. The precision and recall metrics have shown that the SNI AFL approach could perform better than all other approaches. The values of precision and recall exceeded those acquired with other approaches in all test cases.

Despite the good quality of retrieved traces, SNI AFL has a serious drawback: the approach is very resource intensive and not scalable. The main problem occurs when generating the pseudo traces from the BRCG. During that task, the system needs to keep too much information in the memory at the same time in order to avoid duplicate traces. For big systems, this approach might be unrealizable on a desktop computer and would need a power computer with more memory and calculation capacities. We should, nevertheless, admit that the quality of traces acquired from this combined technique surpassed the quality of individual techniques separately.

2.3.4.2 CERBERUS

CERBERUS [10] is another example of hybrid traceability techniques. It combines all three types: static, dynamic, and textual technique. The static technique is realized by the *prune dependency analysis* which has been developed by Antoniol et al. [10] for programs' structural analysis. The dynamic technique and textual technique are based respectively on execution tracing analysis and information retrieval.

A *prune dependency* relationship is defined between a program element and a requirement (concern) when following rule is satisfied: “A program element is relevant to a concern if it should be removed, or otherwise altered, when the concern is pruned” [10]. In other words, the prune dependency relationship between a requirement “ r ” and a code element “ e_r ” exists if removing the requirement “ r ” would induce the removal or change of the element “ e_r ”.

The *prune dependency analysis* (PDA) is supposed to find such prune dependency relationships in a program. As input, PDA expects a set of *initial relevant elements* to a requirement and based on their relationships in the proposed program, PDA infers *additional relevant element*. The initial set could be manually created by a software analyst or automatically retrieved by another technique. Formally, PDA’s input is defined as a tuple (G, R, A) for each requirement. G is a directed graph that expresses the structure of code elements in the program and three types of dependencies between them: references dependencies (e.g. calling references), contains dependencies (e.g. class `Multiplier` contains a method `product()`), and inherits-from dependencies (e.g. class inheritance in java). The graph G is common for all requirements and will be created only once. R is called the *removal set* and it contains the set of initial relevant elements that have to be removed when a given requirement is pruned. A is the *alter set*. It will be filled with the elements that should be changed when pruning the given requirement. In contrast to graph G , R and A are depending on a specific requirement. Basically, the alter set A will be filled during the process and the user does not need to define it as the case with the removal set R .

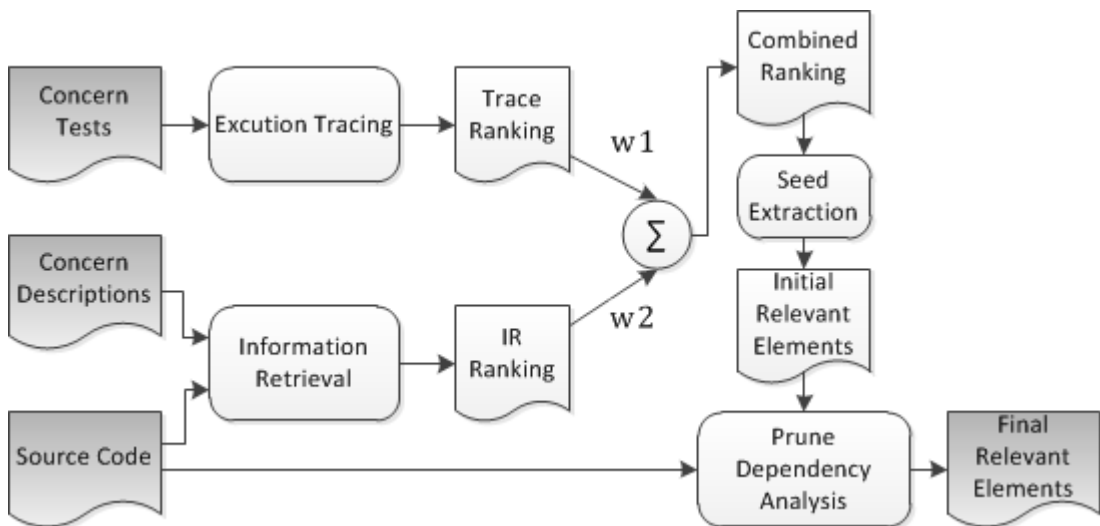


Figure 2.8: Overview of the CERBERUS approach [10]

The prune dependency analysis runs multiple times over the given graph G and simulates the removal of each requirement separately. The process starts from the predefined

initial relevant elements and analyzes their neighboring elements recursively. Assuming that the initial relevant elements are correct, the process classifies neighboring code elements into removal or alter set. The process applies following checks on visited code elements:

1. If the current element has dependencies only on elements in the removal set, it will be inserted in the same set and will be considered as an element that should be removed when pruning the requirement.
2. If the element has dependencies on elements in the removal set as well as other elements, it will be considered as a shared element with other requirements. It might be necessary to change it when pruning the requirement. Therefore, these elements are inserted into the alter set.
3. If no direct dependency on elements in the removal set was detected, the element will be ignored.

The final relevant elements are the union of both removal and alter sets. These elements are supposed to be the code elements implementing the requirement in question. Figure 2.8 shows a detailed overview of the CERBERUS approach. The initial relevant elements are extracted using execution tracing (dynamic) and information retrieval (textual) techniques. Each technique will be performed separately and deliver a ranked list of possible relevant code elements. Then, results from both approaches are combined into a single ranked list of initial relevant elements that represents the removal set R for the prune dependency analysis. At the same time, the graph G will be automatically generated from the program source code. As soon as G and R are computed for a requirement, PDA will analyze G in order to determine the final relevant elements.

The dynamic analysis applied in this approach is very similar to “execution slices” (see section 2.3.2 on page 17). The execution slices technique was extended with a ranking mechanism in order to adopt it in this approach. Multiple metrics are applied to create a reliable raking mechanism. For example:

- *Software Reconnaissance* (SR) [26] of a code element is a Boolean metric that takes the value 1 if the element is triggered (called) at runtime by a unique requirement and 0 otherwise.
- *Dynamic Feature Traces* (DFT) [4] of a requirement is the ratio of elements triggered when executing that requirement from total number of elements triggered at runtime.

- *Scenario-Based Probabilistic Ranking* (SPR) [12] of a code element is the ratio of unit tests of a specific requirement triggering it from total unit tests executing it (including unit tests of other requirements).
- *Element Frequency-Inverse Concern Frequency* (EF-ICF) [10] is similar to *tf-idf* (term frequency and inverse document frequency) in information retrieval. EF-ICF is defined as:

$$IF - ICF = \frac{\#elem\ triggered\ by\ the\ req}{total\ \#elem\ triggered} \times \log \left(\frac{\#reqs\ triggering\ any\ elem}{\#reqs\ triggering\ current\ elem} \right) \quad (2.6)$$

The CERBERUS approach was verified with Rhino⁵ which is an open-source implementation of JavaScript written entirely in Java. Rhino is implementing the *ECMAScript international standard*⁶ (ECMA 262 specification). Antoniol et al. [10] considered it as a requirements documentation for Rhino, and decided to retrieve the Rhino requirements definition from that standard specification. The ECMA specification was very well organized. Each function was described in details in a separate section. Antoniol et al. [10] proposed taking each *leaf* sections (a sections without further subsections) from the specification as a requirement description. The IR application used these descriptions as queries over the set of code elements which are considered like a document set. The relevance elements are identified and ranked using a conventional vector space model (see section 2.3.3.2 on page 22).

The initial goal for combined techniques is improving the quality of retrieved traces. It is important to measure the gained quality over using different techniques individually. In addition to precision and recall metrics, the effectiveness of CERBERUS was measured with the weighted harmonic mean of precision and recall, namely f-measure:

$$F = \frac{2 \cdot precision \cdot recall}{(precision + recall)} \quad (2.7)$$

Instead of using precision and recall, the f-measure summarizes the effectiveness of the system in one single metric. The f-measure is, however, still specific to single requirements. Antoniol et al. [10] considered computing the *mean recall*, *mean precision*, and *mean f-measure* metrics that represent the respective averages of recall, precision and f-measure among all requirements in Rhino. They also tested all different combinations of the three applied techniques (Information retrieval, dynamic analysis, and PDA). The test cases included single techniques and combined ones (two by two). This is necessary to compare the quality acquired from CERBERUS with other possible techniques. As expected, The hybrid techniques did nevertheless perform better than single

⁵<http://www.mozilla.org/rhino>

⁶<http://www.ecma-international.org/publications/standards/Ecma-262.htm>

technique. CERBERUS did outperform almost all other techniques individually and combined. The only exception was with the combination of IR and PDA. It has always had a very close effectiveness to that of CERBERUS and it did even outperform CERBERUS in few cases.

2.4 Trace Validation

While the relationships between traces and calling relationships in the code have been explored in the past [22, 10], they have never been exploited together with potentially erroneous traces for validation. Indeed, the strongest motivation for our work comes from a study by Kong et al. [18] who conducted an experiment about manual trace links validation. Many software analysts participated in validate existing trace links and retrieve missing ones if possible. The participant analysts were senior and graduate students in computer science and software engineering from two universities: the University of Kentucky and Cal Poly University. Each of the participants had to validate a number of trace links from given Requirements Trace Matrices (RTM) which were retrieved automatically using Information Retrieval.

The validation of traces is simply the task of checking traces for correctness. The participant analysts have to look at each available trace and decide whether to confirm or to reject it. A basic tool support was also provided in order to record the behavior of each analyst during the experiment. The tool, called RETRO, tracked multiple user actions and logged them into an external file for later analysis. The logs included various detailed information, such as: time stamp, action type (keyword search, selection...)

In order to evaluate the decisions taken by each participant, a research team generated in advance a *golden standard RTM* which represents the best accurate RTM. The quality of an RTM is usually measured using Precision and Recall. Higher precision means a higher number of traces in the RTM are correct. A higher recall means that the RTM contains a bigger set of the possible traces. In our case, we assume that the golden standard RTM is our “yardstick” with 100% precision and 100% recall. The accuracy of validated RTMs will be measured relatively to it.

The experiment [18] was conducted with 13 participants. Each of them had to validate a different candidate RTM with a different accuracy levels (different precision and recall). The candidate RTMs are generated in advance. Each RTM had a certain number of correct and incorrect traces. Meanwhile, participants worked on rejecting incorrect traces and confirming correct ones, the supporting tool recorded the details of analyst’s work in external log files. The principal idea of the experiment is to understand the

human behavior during validation task by analyzing the recorded logs. The log analysis helped identifying multiple strategies used by the participants to accomplish their task. For example: some participants did most of the time review all links before taking any decisions; other participants took decisions about traces that seemed to be obvious before reviewed all traces; some others consulted available traces first, then searched for keywords before taking their first decision. . . The analysis of these logs should help answering important research questions, such as: How much source elements did a participant read before taking a decision about the correctness of a trace? How much time was spent searching for links? At which stage of the task did the analyst face the most difficulty?

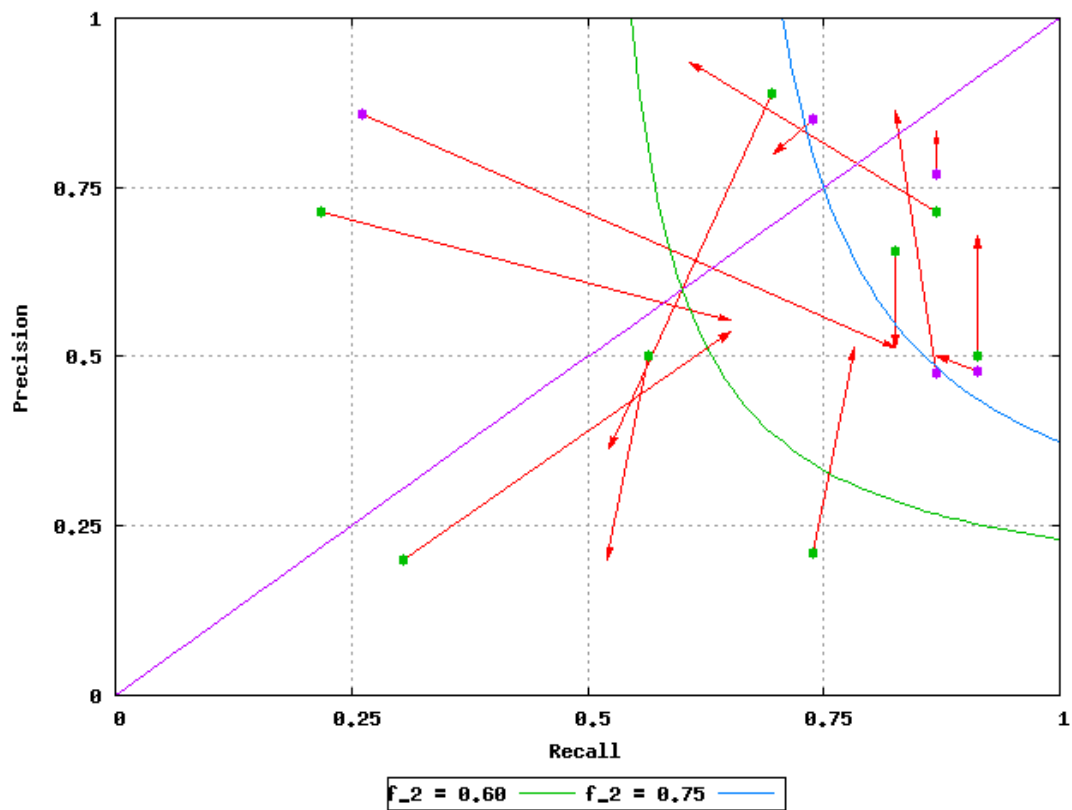


Figure 2.9: Performance of the 13 study participants plotted in the precision-recall space [18]

Logs analysis revealed few important observations. Six of the participants started slowly with a number of incorrect decisions before they stopped making mistakes after a certain time. In average, they spent at least 20 minutes until they started identifying correct traces. Half of the participants were reviewing all links during the earlier stage of the task, which explains the delay in reaching good decisions. Other four participants were able to locate correct traces earlier in the experiment making only few incorrect decisions throughout the task. Compared to the other participants, these four analysts

were able to get past the hurdle quickly and were able to go faster through candidate traces. A period of “tiredness” was recognized in the work of two other participants. They started making good decisions before committing multiple mistakes in a row, but they recovered after a while and were again able to take correct decisions.

Figure 2.9 shows the performance of all participants plotted as vectors in a precision-recall space. Each vector starts at the initial accuracy (recall, precision) of the candidate RTM before the validation experiment. It ends at the resulting accuracy after validating it by a participant. If the manual validation would improve the quality of the matrices, we would see most of the vectors pointing towards the upper-right corner of the space where the precision and recall are at their best values ($recall = 1$, $precision = 1$). The participants validating RTMs with poor initial accuracy tended to deliver better quality RTMs. E.g. a user, who started with a RTM that had only 7 correct traces out of 35 candidate traces ($recall = 0.304$, $precision = 0.200$), submitted a RTM with 15 correct traces out of 28 total correct traces ($recall = 0.536$, $precision = 0.625$). He significantly improved the RTM quality by 34.8% in recall and 33.6% in precision. Unfortunately, this has not been the case with all participants. Those who started with a good quality RTM ended up making them worse. Few of them succeeded in improving the precision or the recall at the expense of the other. This can be seen in figure 2.9 on hand of vectors pointing from upper-right (best quality) corner to the bottom-left (worst quality) corner. There are also participants who’ve got very small RTMs at the start. So, they finished very quickly validating the available traces and started looking for missing traces with the belief that the given RTM is incomplete. The newly added traces were in most of the cases wrong and instead of improving the quality of the RTM, they ended up deteriorating the initial quality.

This experiment creates a fundamental understanding for manual traces’ validation. We learned that manual validation does not necessarily improve the quality of traces. Many participant analysts did not succeed in recognizing erroneous traces and they did even deliver other wrong traces. The quality of traces was always depending on the quality of initially available traces. When starting with a low quality RTM, participants did perform well and improved the overall quality of traces. But when they started with high quality RTM, the results lost remarkably in quality. The analysts delivered, nonetheless, more trustworthy traces as long as they are exclusively working on a single task: either trace recovery or race validation. As soon as they start switching from one to another, they begin taking wrong decisions.

Kong et al. [18] noted also an interesting correlation between code complexity and quality of traces. This experiment confirmed a previous observation made by Egyed et al [11] saying that: “When an analyst spends very little time on a link they tend to make the correct decision. On difficult links, where the analyst struggles to make a decision,

they frequently commit to the incorrect decision” [18]. Both studies were performed independently, what emphasizes the conclusion that spending more time on some code locations does not necessarily deliver better traces. It is rather the complexity of the code that makes it understandable and easy to trace or not.

The experiment reveals important aspects of manual traces validation. There are however few concerns that are worth to discussing. The experiment could be biased due to the Hawthorne effect: ”It is a form of reactivity whereby subjects improve or modify an aspect of their behavior being experimentally measured simply in response to the fact that they are being studied.” It is impossible to confirm that the behaviors observed during the experiment are covering all possible human behaviors during a manual traces validation. The experiment could also be influenced by other human factors, such as: familiarity with the code, experience with different projects... The observed aspects are, nevertheless, very realistic, and hence we consider them as a good simulation for real world behaviors.

Chapter 3

Hypothesis of Surroundedness

Intuitively, we think that methods implementing a given requirement are somehow connected in their implementations, and hence traces are not random occurrences in the source code. If a requirement is implemented in multiple methods, it is, indeed, intuitive to think that methods implementing that requirement are likely to be calling each other when executing that requirement. Thus, there should be some kind of relationship between the different methods sharing the same requirement that we expect to see in a call graph. e.g. we expect to see the callers and callees of a given method are more likely related to the same requirement.

For a program, call graph could be generated in multiple ways. On the one side, there are static call graphs that are generated out of the structural properties of a program, such as: concern graph (see section 2.3.1.1 on page 13), or abstract system dependence graph (see section 2.3.1.2 on page 15)... On the other side, there are dynamic call graphs that are generated at runtime depicting executed call relationships between code elements. In this thesis, we decided to use dynamic call graphs due to multiple reasons. Compared to static graphs, a dynamic call graph would contain only needed code and thus, it truncates implicitly dead code and reduces the size of the graph which would improve the performance of any algorithm running over the graph structure.

In this chapter, we will present the Requirement Call Graph (RCG) data structure which helped us analyzing and verifying our assumptions and all related issues. But before, we will introduce the case study systems that we used for the investigation of our hypothesis which we will present in the last section of this chapter.

3.1 Case Studies

We are basing our observations on the evaluation of four third-party software systems (between 3-72 KLOC) and 59 randomly chosen requirements. The four open source projects of different sizes and different domains were:

- **VideoOnDemand** (VoD) is a MPEG-Decoder library that provide a server-client architecture for video streaming. It supports basic functions, such as: starting and stopping a video stream. This project has been reverse engineered by professional software developers. The resulting source code is a relatively small with only 3.6 KLOC. The developers did also a manual traces recovery and provided traces for 14 sample requirements.
- **Chess** ⁷ is a simple Swing implementation of the chess game. It runs locally like any other java application. The user can start a game against the computer or against another user. The option for online gaming is not yet introduced in the version that we are using for our experiment. The developer of Chess provided traces of the main 8 requirements implemented in the test version.
- **GanttProject** ⁸ is an open source project planning and tracking tool. It allows the user to create project tasks in form of charts diagrams and set dependencies among them. GanttProject, or briefly Gantt, provides basic support functions for project analysis, such as: computing the start and finish dates of projects and individual tasks, computing critical path for responsible tasks for delays, and visualizing bottleneck tasks in the project. It also includes functions for human resources management. The system is quite large, consisting of 41 KLOC Java code distributed over more than 500 classes and 3000 methods. We selected 16 example requirements to be investigated in this work.
- **jHotDraw** ⁹ (JHD) is a java framework for drawing technical and structured Graphics. The project was originally an exercise for using design patterns, which makes it a well-designed project and a good candidate for investigation. The program has a simple user interface (UI), as usual in painting programs, that allows to select, draw, and modify graphical elements. This is our biggest test project with 72 KLOC and 21 requirements.

The basic characteristics of these projects are depicted in Table 3.1. All of them are implemented in Java. These systems were chosen, in part, because of the availability

⁷<http://www.java-chess.de/>

⁸<http://ganttproject.biz/>

⁹<http://www.jhotdraw.org/>

of high quality requirements-to-code traces. Particularly for the larger two systems (Gantt and JHD), the institute for Systems Engineering and Automation (SEA) provided monetary compensation to lead software engineers in charge of building these systems to ensure that the RTMs were done well.

	VoD	Chess	Gantt	JHotDraw
Programming Language	Java	Java	Java	Java
Size in KLOC	3.6	7.2	41	72
Nr. of Executed Methods	173	416	2603	1768
Nr. of Sample Requirements	14	8	16	21
Golden Standard: Size of RTM	2422	3328	41648	37128

Table 3.1: Properties of case study projects.

The available traces are of high quality and therefore we consider them as the golden standard for our systems. We recall that a golden standard RTM is the best possible RTM assumed to be completely accurate. Having these RTMs as golden standards was crucial for two reasons:

1. to conduct our analysis on correct real world examples in order to get trustworthy observations.
2. to test our approach’s ability to cope with incorrectness by randomly seeding errors into these RTMs.

	<i>R1: Play Single Move</i>	<i>R2: Show Score</i>
Game	t	t
Board	n	n
Player	t	n
Control	t	t
Piece	n	n
Pawn	n	n

Table 3.2: Excerpt of RTM from Chess system.

These traces were captured in form of Requirements Trace Matrices (RTM). Each RTM contains $m \times n$ cells where m is the number of code elements and n is the number of requirements. Table 3.2 depicts a tiny excerpt of such a RTM for the Chess system. Each cell indicates either a trace ('t') or no-trace ('n') as was discussed previously. A trace in a cell implies that the given code element (row - a class in this case) implements the given requirement (column). Vice-versa, a no-trace in a cell implies that the code element is not implementing the requirement. A row in the RTM thus identifies all requirements that a method implements (or by negation not

implements). Each column identifies all methods that implement a requirement (or by negation does not implement). Typically, there is a many-to-many relationship in how requirements are implemented in code.

3.2 Requirement Call Graph

As was mentioned in the introduction, our approach uses calling relationships among code elements to help validate the correctness of the given requirements-to-code traces. How these calling relationships are created is not important in principle. But since static analysis of source code is typically not able to identify method calls completely, we chose to observe them dynamically by execution the system. The Java JDK provides an easy and reliable interface for recording method calls at runtime . No special prerequisites were necessary for the execution of the systems. We simply tested the systems as exhaustively as could be. The testing was not limited to the sample requirements nor was it attempted to test the requirements individually. This was done to avoid any bias towards particular requirements or usage scenarios. Though, we believe that structuring the tests towards individual requirements could further improve the quality of our approach, which is future work . There are standard tools available for recording execution logs during program execution. Such execution logs are in essence tree-like structures where each executing thread would be represented by a separate tree, where methods are nodes and calling relationships are edges (edges are in fact implied through the hierarchical embedding of methods inside other methods). The root of a *call tree* is the starting method of any given thread (e.g. the `main()` method is one such root). Each method could have one or multiple representing nodes depending on how often it has been called during the execution.



Figure 3.1: Caller and Callee relationships

The sizes of the resulting execution logs were huge and challenging to process directly. We thus reduced these call trees into call graphs where methods are represented by single, unique nodes. All available calls in the call tree are then copied to the call graph such that duplicate calls are merged. The result is a much smaller graph structure (averagely about 5% the size of the original call tree), containing as many nodes as there are methods and as many edges as there are distinct kinds of method calls. From the call graph, it can no longer be established how often methods call one another,

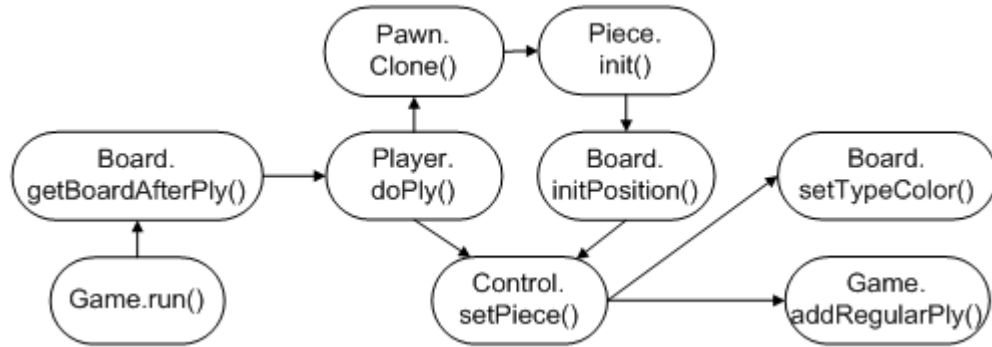


Figure 3.2: Excerpt of Call Graph for Chess System

however, this information is not important for our approach. We distinguish between *callee* methods which are called by a method, and *caller* methods which are calling a method. When we say $M1$ is a callee of $M2$, it means that $M2$ is calling $M1$. Vice-Versa for caller relationship: If we say $M3$ is a caller of $M2$, it means that $M3$ is calling $M2$ (see Figure 3.1).

The call graph also combines multiple threads' call trees into a single graph if they share at least one method. In addition to being much more compact, the call graph has the additional advantage that each node can be matched exactly to one row in the RTM (because a node in the graph represents a method as does a row in the RTM). Deriving the call graphs out of a call tree is a simple process (details omitted for brevity). It is important to note that neither the call tree nor the call graph are connected to the RTM. Both can and were established independently of the RTM input (the call graphs were in fact created in course of this project and are presumed correct). Figure 3.2 illustrates an excerpt of the call graph generated from Chess call tree.

We derived a Requirement Call Graphs (RCG) by associating the call graph information with a requirement from the RTM. Each requirement will be presented by a single RCG. In the RCG, each node will be labeled with the trace link corresponding to its method in the requirement trace matrix. Trace method nodes will be labeled with a “t” and no-trace method nodes with an “n”. For each requirement, we get an RCG that contains both: the call graph information and the trace/no-trace links from the RTM. Each requirement has its RCG where the method nodes are labeled with “t” and “n”. The derivation of such a graph is straightforward and does not require much effort. The operation is as simple as associating indexed data from two data structure into one single structure. Figure 3.3 shows an example requirement call graph derived from the combination of the call graph in Figure 3.2 and the requirement $R1$ from the RTM in Table 3.2. Additionally, we colored the trace methods in green and the no-trace ones in orange to recognize the different labels more easily.

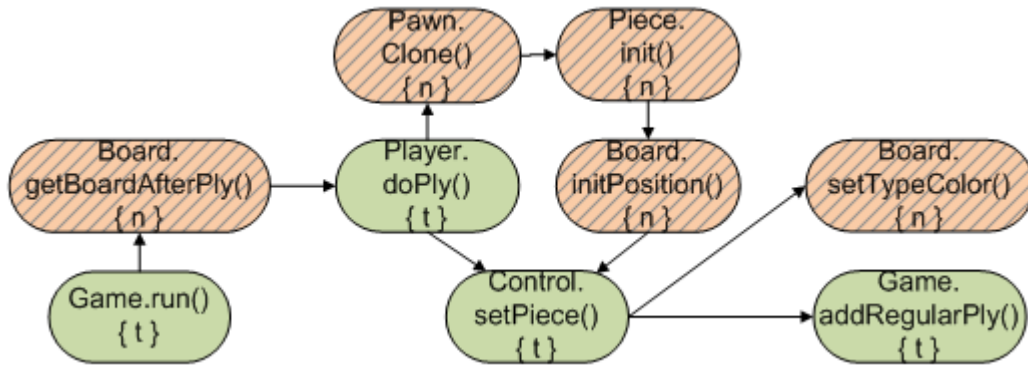


Figure 3.3: Requirement call graph for Chess System

3.3 Hypothesis

As mentioned previously, we intuitively think that there is a correlation between the trace/no-trace relationships of methods to requirement and the calling relationships. We expect a tracing method to be surrounded with other tracing methods. In other words, the callers and callees of a method should reflect the trace relationship of a method. When a method is said to be tracing to a requirement, we should also find callers and callees methods tracing to the same requirement. Analogically, when a method is said to be not-tracing to a requirement, we should find callers and callees of that method which are not tracing too. Our hypothesis is that we can use such correlation to identify correct (trace and no-trace) links from wrong ones.

Chapter 4

Observations

Although our hypothesis seems to be very intuitive and realistic, we have to prove that such a correlation between trace/no-trace links and calling relationships exists. This is, of course, the most challenging part of this work, since we have to do experimental research on case study systems in order to confirm our assumptions and then deduce how to use this knowledge to validate traceability links. In this chapter, we will explain the observations that we were able to capture on case study systems. We identify properties that could be used for the recognition of correct traces from wrong ones. *Understanding these properties is important for understanding why our approach and particularly its simple patterns are highly effective.*

4.1 Clustering

On first inspection, the relationship between any given requirement and code is not straightforward. If some method A implements a requirement R and method A calls method B then one of the following two situations applies:

1. method B implements a service required by method A and, by implication, implements requirement R also
2. method B implements another requirement that is meant to coincide when method A occurs. By implication, method B then does not implement requirement R.

Both situations should be visible in the requirement call graph (RCG). Considering the colors that trace links (green) and no-trace links (orange) have, the situation (1) would induce a very clear colors distribution where a single group of green nodes is visible and all other nodes are orange. In the situation (2), however, we would get a graph with

a lot of small groups of green nodes which are separated from each other with orange nodes. The distance between the single groups might vary from one case to another, but most of them should be approximately in the same part of the graph where the requirement in question is supposed to be executed.

Clustering investigates whether all callers of a method and/or all its callees implement the same requirements as the method itself. Thus, clustering investigates whether situation (1) is more likely than situation (2) - perhaps a pattern to be exploited. Since it is not possible to reason about this theoretically (both situations above appear reasonable and probably both should occur), we investigated clustering in context of the four case study systems and their respective golden standards. Table 4.1 depicts the empirical results.

	Trace Clustering		No-Trace Clustering	
	Caller = 't'	Callee = 't'	Caller = 'n'	Callee = 'n'
VoD	54.33%	57.31%	91.04%	90.00%
Chess	74.24%	76.39%	81.09%	79.26%
Gantt	54.49%	45.21%	88.82%	92.02%
JHotDraw	31.96%	34.75%	95.03%	94.40%

Table 4.1: Likelihood of clustering

We see significant differences between the various systems but also between methods that trace to a given requirement and those that do not. If a method traces to a given requirement then, depending on the system, it is 34-76% likely that its callee also traces to that requirement (column 3 in table 4.1) or 31-74% likely that its caller traces to that requirement (column 2 in table 4.1). For example, in case of Gantt, only 45% of the callees trace to the same requirement the caller is tracing to. Notice that the likelihoods change drastically for 'no trace' observations. If a method does not trace to a given requirement then it is 79-94% likely that its callee also does not trace to that requirement (column 5 in the table 4.1). The higher percentage is explained in the fact that traces are rare compared to no traces (at least for the 59 requirements we randomly selected). Much of the code does not trace to individual requirements. Therefore, some random code that is known to not trace to a requirement is likely surrounded by other code that does not trace to that requirement. The exception to this is the Chess system where the requirements covered larger areas of the code and correspondingly we see likelihoods that are more balanced.

Obviously, trace are not building clusters in the way that we were expecting it. Even graphically, we did not see the groups of green nodes as explained at the beginning of this section. The tracing nodes were rather connected to each other without building a clear green group. Next, we will investigate the property of connectedness between traces and no-traces.

4.2 Connectedness

While not all callers and callees of a method seem to relate to the same requirements, it seems likely that at least one caller or callee relates to the same requirement. The connectedness metric represents the percentage of methods that are directly connected to at least one other method implementing the same requirement (where connected means having a caller or callee).

Table 4.2 depicts the connectedness of trace methods on the four systems. Here, we distinguish between *inner nodes* and *leaves*. Methods are so-called inner-nodes when they have at least one caller and one callee. The connectedness of inner nodes can be evaluated on both the caller and callee side (is there at least one caller and/or one callee?). A method is a leaf node if it does not have callee methods and thus, the connectedness can be evaluated on the caller side only. There are also root nodes but these are extremely rare. There should be a root node per thread and often there is only one root node which is the `main()` method in Java. We ignore root nodes in this work because of their limited number.

	Trace Connectedness		No-Trace Connectedness	
	Caller = 't'	Callee = 't'	Caller = 'n'	Callee = 'n'
VoD	88.50%	59.00%	98.50%	93.30%
Chess	99.39%	93.18%	98.98%	92.23%
Gantt	94.31%	76.48%	99.59%	91.99%
JHotDraw	90.15%	72.16%	99.87%	97.77%

Table 4.2: Percentage of connected trace/no-trace methods

For inner-nodes, we have a very high connectedness of 88-99%. Leaves do not perform as well as inner-nodes but are still 59-97% connected. We can see that no-trace methods show higher connectedness values and the no-trace leaves perform nearly as well as no-trace inner-nodes. The high number of no-trace methods compared to trace ones could explain this again.

The observations of trace/no trace *connectedness* emphasized our assumption that methods' trace are highly interconnected. This behavior is more visible for inner-nodes but also strong for leaf nodes. Higher connectedness implies that a method that traces to a requirement is very likely to find another neighboring method that also traces to that requirement.

4.3 Requirement Call Chain

However, are these connected methods isolated or do they form a chain where all (or most) methods that implement a given requirement are connected directly or indirectly with other methods that implement the same requirement?

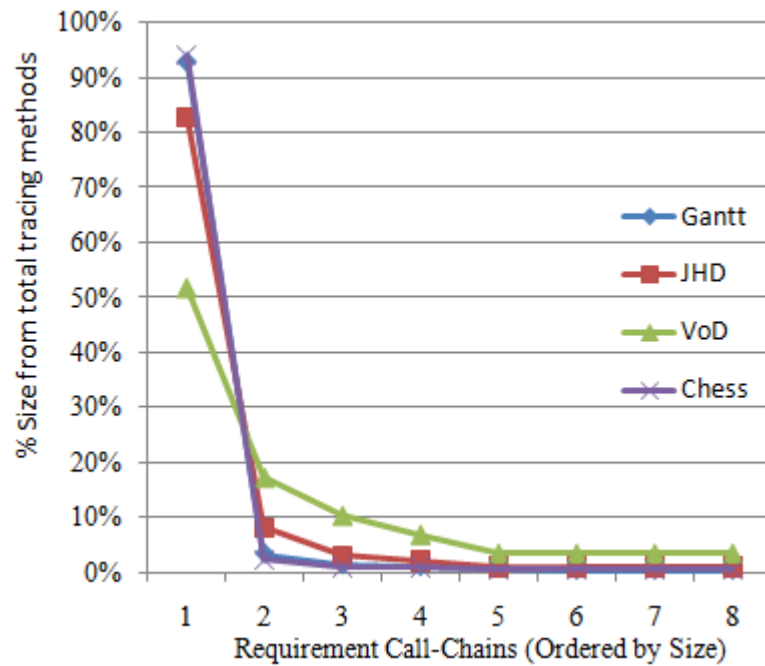


Figure 4.1: Most methods implementing a given requirement are connected directly or indirectly by method calls

To observe this, we grouped methods into regions if they trace to the same requirements and are connected via calling relationships. All 59 requirements we studied were implemented in multiple methods and we surprisingly found that all requirements exhibited the same effect: there is usually one big region of connected methods which implement a given requirement and several small remaining regions that for the most times, do not contain more than a single method. Figure 4.1 shows the distribution of tracing methods over different regions ordered by size (average group size over all requirements in each system).

It is important to note that these regions of methods typically form a long chain of connected methods. These *call chains* have different lengths depending on the size of the requirement. The methods in the call chain have different properties. Few of them have multiple connections. The call chain is “thicker” in some locations (where multiple callers or callees trace to the same requirement). In other places, the call

chain is “thin” (where only a single caller and/or a single callee traces to the same requirement). The figure 4.2 shows an excerpt of a Chess RCG indicating how a call chain could be connected among all other nodes.

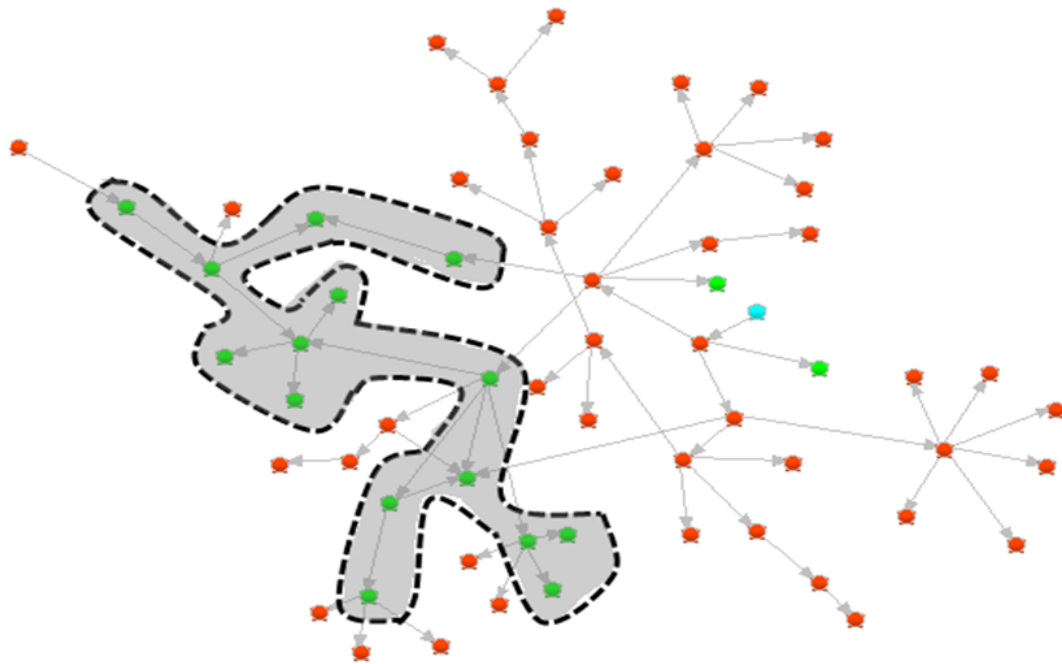


Figure 4.2: Call chain from a Chess RCG

Clearly, there is a strong indication that calling relationships correlate with traces but they are not exactly alike. Next, we present patterns that exploit these basic connectedness properties of traces.

4.4 Patterns

By watching the colors distribution in the RCGs from different case study systems, we were able to recognize some kind of patterns in the graph. We decided to identify those patterns first and then find their correlation with the correctness of traces/no-traces. In this section, we will explain the different patterns that we were able to derive and the meaning of each of them to our approach.

4.4.1 Surrounding Patterns

We define the principle of *surroundedness* as meaning that a method is likely to trace to a requirements if at least one caller method and at least one callee method trace to that requirement. The same is true for no trace. This principle is based on the observation we made above. Yet, it is important to note that this observation is based on the investigation of individual requirements. At no point did we mix requirements which manifested itself in the simple terminology we used thus far: a 't' for tracing to an individual requirement and an 'n' for not tracing to it. Throughout the remainder of the work, we continue to adhere to this strict separation.

A Requirement Call Graph (RCG) combines the call graph discussed earlier with the trace information of a single requirement. Each requirement has its own RCG with all RCGs sharing an identical structure but different labels ('t' and 'n'). In the RCG, each node will be labeled with the trace link corresponding to its method in the requirement column in the RTM. Trace method nodes will be labeled with a "t" and no-trace method nodes with an "n".

We speak of a *t-surrounding* method as having both a caller and a callee who trace to the same given requirements (caller='t' and callee='t' - hence surrounded). Similarly, we speak of a *n-surrounding* method as having both a caller and a callee who do not trace to the same given requirement (= 'n'). We denote these two patterns as " $t \succ ? \succ t$ " and " $n \succ ? \succ n$ " where "?" stands for some random method, the arrow (\succ) beforehand refers to the caller and the arrow thereafter refers to the callee (\succ). The caller or callee is either required to be a trace 't' or no trace 'n'. Hence, " $t \succ ? \succ t$ " represents the *t-surrounding pattern* and " $n \succ ? \succ n$ " represents the *n-surrounding pattern* (see Figure 4.3 for an example of t-surrounding and n-surrounding in context of the Chess system).

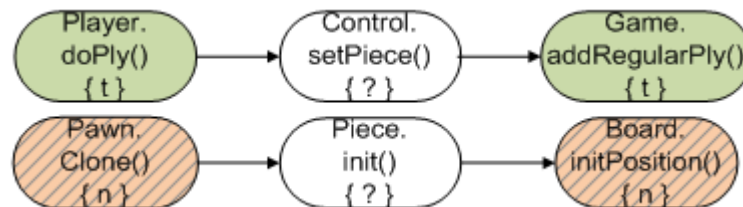


Figure 4.3: Examples of " $t \succ ? \succ t$ " and " $n \succ ? \succ n$ " patterns.

As implied by the principle of surroundedness, the t-surrounding pattern implies that the given method (?) should be a trace while the n-surrounding pattern implies that the given method should be a no trace. Table 4.3 depicts these likelihoods, again

in context of the four case study systems. For example, the likelihood for a random method that exhibits t-surrounding tracing to the same requirement ($t \succ ? \succ t$ such that $?=t$) is between 61-96% depending on system. The likelihood for a random method that exhibits n-surrounding to not trace to a given requirement ($n \succ ? \succ n$ such that $?=n$) is between 86-96%. This likelihood is computed from the golden standard RTMs of the four sample systems. For $t \succ ? \succ t$ being $?=t$, the likelihood is computed by counting the patterns “ $t \succ t \succ t$ ” and “ $t \succ n \succ t$ ” in the RCG where $t \succ t \succ t$ stands for three methods calling each other in a sequence and all methods tracing to the same requirement and $t \succ n \succ t$ with a similar sequence except that the middle method does not trace to the given requirement:

$$\text{likelihood of } t \succ ? \succ t \text{ where } (? = t) := \frac{\#(t \succ t \succ t)}{\#(t \succ t \succ t) + \#(t \succ n \succ t)} \quad (4.1)$$

	VoD	Chess	Gantt	JHotDraw
$t \succ ? (? = t)$	76.4%	45.2%	34.8%	57.3%
$n \succ ? (? = n)$	79.3%	92.0%	94.4%	90.0%
$t \succ ? \succ t (? = t)$	61.5%	96.0%	73.3%	61.8%
$n \succ ? \succ n (? = n)$	93.9%	86.4%	93.9%	96.3%

Table 4.3: Likelihood of calling relationship patterns

Unfortunately, The t-surrounding and n-surrounding patterns are useless for leaf methods because these have callers only (no callees). Since leaf methods amount to roughly 50% of the nodes in the RTM, we cannot ignore them. In the absence of a callee, we define:

- a t-surrounding pattern for a leaf node as having a caller that trace to a given requirement ($t \succ ?$). Table 4.3 reveals that this pattern resolves between 34-76% to a 't' (less than the t-surrounding pattern for inner nodes).
- a n-surrounding pattern for a leaf node as having a caller that does not trace to a given requirement. This pattern is 79-94% likely to resolve to a 'n' as Table 4.3 reveals.

It is noteworthy that the t-surrounding and n-surrounding patterns are small patterns involving two calls and three methods only (the random method, its caller and its callee). Since there are potentially many callers and callees for any given method, these simple patterns must be combined. We will see further in this work that by considering these simple patterns in combination, we are able to compute expected traces/no traces with >90% correctness for most cells of the RTM. Before we can discuss

this combination, we must first discuss failure to identify surroundedness (boundary patterns) and the concept of dominance between patterns.

4.4.2 Boundary Patterns

Not every pattern encountered must be a t- or n-surrounding pattern. As was defined previously, requirements appear to be implemented in call chains of connected methods in form of call chains. These regions have boundaries where a method not implementing a given requirement may call a method implementing one or vice versa. Exactly along such boundaries, we have difficulty identifying surroundedness.

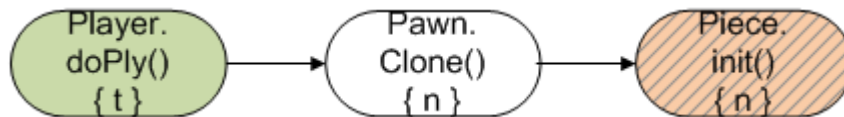


Figure 4.4: A boundary pattern “ $t \succ ? \succ n$ ”

We find such boundary patterns at entry and exit points to and from regions. We speak of a boundary pattern as having a caller that traces to a given requirement while the callee does not (i.e. “ $t \succ ? \succ n$ ” as depicted in Figure 4.4); or as having a caller that does not trace to a given requirement while the callee does ($n \succ ? \succ t$).

It is not possible to decide whether the given method (‘?’) in a boundary pattern should trace to a requirement or not. It is easy to see that the given method could be either just outside the requirement region (in which case the $? = n$) or just inside said region (in which case the $? = t$). Empirical evaluation on the four case study systems confirmed this. These boundary patterns are thus failure patterns where no expected trace can be computed.

A similar argument could be made for leaf nodes. We consider leaves that are called by multiple methods with different trace links as boundary locations. In other words, a leaf method having mixed caller links, some of which are traces (‘t’) and others are no-trace (‘n’) are considered as boundary patterns. Such a case occurs when a leaf method is required by multiple caller methods, yet these methods do not implement the same requirement(s). In context of leaf nodes, boundary methods are often general purpose methods.

Since our approach cannot decide on boundary patterns, it should fail in many cases. However, since methods are often surrounded by multiple callers and callees, boundary

pattern commonly occur together with t- and/or n-surrounding patterns, which can be decided upon. The following first discusses dominance in cases where t- and n-surrounded patterns occur together. We then discuss scenarios in which boundary patterns occur together with t- and n-surrounding patterns.

Chapter 5

Automated Trace Validation Approach

The surrounding concept that we have just introduced in previous section about the patterns leads us to create rules for estimating trace links. Apart from being an inner or a leaf node, we are able to define precise rules for estimating trace values in several cases. On the one side, t-surrounding and n-surrounding patterns provide two different rules which estimate respectively a trace 't' and a no-trace 'n' for a method on which they apply. On the other side, boundary patterns indicates the boundary of a call chain (region) of traces and hence it defines a critical location were our tool fails to generate any estimation.

Unfortunately, there are cases where multiple rules apply at the same time. We discuss in this chapter how to manage those cases in order to select the most appropriate rule. First, we explain the dominance relationship between traces and no-traces that will be also reflected on the t-surrounding and n-surrounding cases. Then, we discuss their relationships to the boundary pattern. And finally, we propose seven categories to organize these rules depending on their combinations.

5.1 T over N Dominance

It is common that a given method has multiple callers and/or multiple callees. It is thus common to have multiple patterns apply to single methods. If these patterns are identical then they confirm each other (e.g., multiple t-surrounding patterns applying to the same method). If different patterns apply to a given method then their combined meaning need to be considered. We previously referred to this as considering the possible combinations of patterns.

It is also common for the t-surrounding and n-surrounding patterns to occur together. This happens when there are at least two callers of which one traces to a given requirement and the other does not; and if there are at least two callees where one traces and the other does not. Figure 5.1 depicts such an example of a combination of t- and n-surroundings on a single method. The one pattern suggests that the given method traces to the given requirement, the other clearly suggests the opposite.

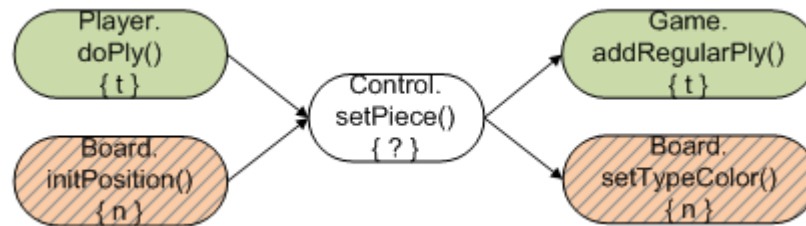


Figure 5.1: Combination of t-surrounding and n-surrounding on method `Control.setPiece()`

This apparent conflict really is none at all. It is common that code implements multiple requirements. If this is the case, the code serves a purpose that supports a trace and another purpose that does not. Indeed, t/n conflicts of this manner are common in the four study systems and, upon investigating this issue, it became obvious that t-surrounding dominates n-surrounding. We define dominates as meaning that in case both t-surrounding and n-surrounding applies to the same method, the t-surrounding wins and the expectation is that the method resolves to 't' (and not 'n' as n-surrounding would have suggested).

The rationale for dominance has to be seen in the context of granularity (an issue of importance to traceability but was ignored thus far). It is said that some code traces to a given requirement if this code implements the requirement in part or full. Traceability is rarely exclusive and as such it is implicitly understood that the code may also implement other requirements. If we think of the code as a class (say a Java class) then the different methods of the class may serve different purposes and implement overlapping or even distinct requirements. Yet, if some of the methods of a class implement a given requirement then we say that the class implements that requirement because it does so in part or full. Much like a method of a class may be involved in an n-surrounding patterns while another method of that class may be involved in a t-surrounding pattern, a single method may be involved in both patterns. If a method implements a requirement in part then the method is said to trace to that requirement and hence *trace dominates the no trace*.

5.2 Boundary patterns vs. t/n-surrounding patterns

The issue of dominance is less obvious in context of boundary patterns and their interactions with t- or n-surrounding patterns. A boundary pattern implies failure to compute an expected trace/no trace, meanwhile the t- and n-surrounding patterns imply success. It would be intuitive to assume that success dominates failure but the issue is more complex. Erroneous 'n's only exist inside requirements regions and are detectable through t-surrounding patterns. Erroneous 't's only exist outside requirements regions and are detectable through n-surrounding patterns. Figure 5.2 (top two rows) depicts these two scenarios.

Boundary patterns, however, occur at the boundary of requirements regions. Unfortunately, it is not always possible to correctly identify this boundary - particularly, which methods should be inside a region and which methods outside. An error along the boundary moves the boundary as can be seen in Figure 5.2 bottom (an erroneous trace method may appear to be outside a requirements region or vice versa).

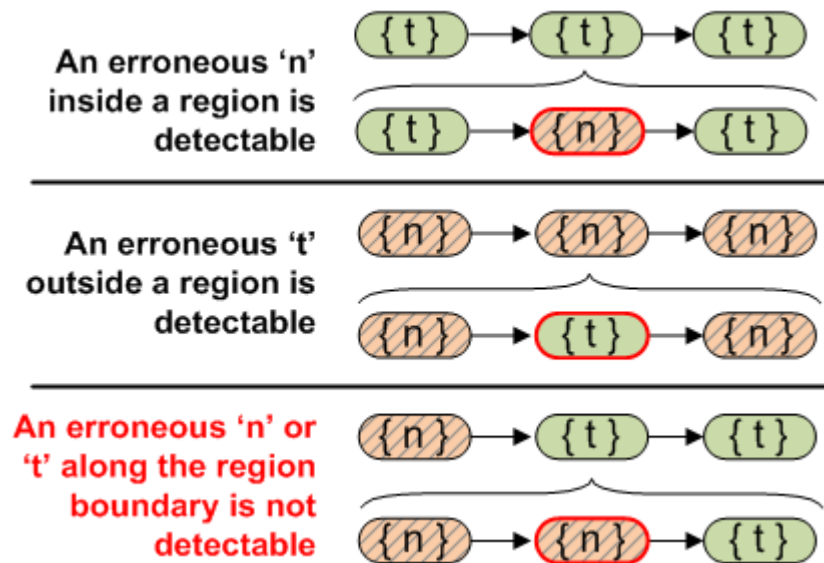


Figure 5.2: Errors inside and outside of requirements regions are detectable, but errors along boundaries of such regions are not.

Simply speaking, boundary patterns denote the limitations of our surroundedness patterns. We should perceive them as warning flags that these areas are problematic and may need more (manual) scrutiny than other areas. We thus distinguish three areas:

- *pure surrounded* nodes where t- or n-surrounding patterns apply only. These situations apply to nodes inside and outside of requirements regions where the absence of boundary patterns implies the highest chance of correctness.
- *mixed surrounded* nodes where both t-/n-surrounding patterns and boundary patterns apply. The situations apply to nodes at the boundary of requirements regions where the presence of the boundary patterns implies a lower chance of correctness. Fortunately, it is common for border patterns to also be t- or n-surrounded, which implies a trace or no trace.
- *boundary* nodes only. These situations apply to nodes that are at boundaries without surroundedness patterns as tie-breakers. These nodes cannot be validated.

Table 5.1 depicts these three areas separately for inner-nodes and leaf nodes (note that mixed surroundedness does not apply to leaf nodes as was discussed previously) as well as for trace and no trace expectations (recall that the likelihoods vary between these). These are labeled as Categories 1-6. Category 7 represents the border which cannot be validated.

Expectation	Inner Nodes		Leaf Nodes	Boundary
	Pure Surrounding	Mixed Surrounding	Pure Surrounding	
no-trace	<i>Category 1:</i> n-surrounded only	<i>Category 3:</i> n-surrounded, boundary	<i>Category 5:</i> n-surrounded	<i>Category 7:</i> (remaining) cannot be validated
trace	<i>Category 2:</i> t-surrounded optional n-surrounded	<i>Category 4:</i> t-surrounded, optional n-surrounded, and boundary	<i>Category 6:</i> t-surrounded	

Table 5.1: Seven categories of surrounding and boundary patterns

5.3 Algorithm

The validation algorithm that we are proposing is an application of the rules discussed in the previous sections. The following snippet is a prose of the validation algorithm:

```

1 getExpectation( Node n, Requirement r ) {
2
3   if ( n.isInner() )
```

```

4     if ((hasNCallerOnly(n, r) & hasTCalleeOnly(n, r))
5         || (hasTCallerOnly(n, r) & hasNCalleeOnly(n, r)))
6         return <CATEGORY_7, FAIL>
7     else if (hasTCallerOnly(n, r) & hasTCalleeOnly(n, r))
8         return <CATEGORY_2, TRACE>
9     else if (hasNCallerOnly(n, r) & hasNCalleeOnly(n, r))
10        return <CATEGORY_1, NO_TRACE>
11    else if (hasTCaller(n, r) & hasTCallee(n, r))
12        return <CATEGORY_4, TRACE>
13    else if (hasNCaller(n, r) & hasNCallee(n, r))
14        return <CATEGORY_3, NO_TRACE>
15
16    else if ( n.isLeaf() )
17        if (hasTCallerOnly(n, r))
18            return <CATEGORY_6, TRACE>
19        else if (hasNCallerOnly(n, r))
20            return <CATEGORY_5, NO_TRACE>
21
22    return <CATEGORY_7, FAIL>
23 }
24
25 validateCell( c RTMCell) {
26     expect = getExpectation(getRCGNode(c.method), c.req)
27     if (c.value = expect)
28         return <SUCCESS, expect.category>
29     else
30         return <ERROR, expect.category>
31 }

```

The validation algorithm is composed of two parts: 1) to compute expectation via `getExpected()` and 2) to compare the expectation with the actual value via `validateCell()`. The algorithm for `validateCell()` is called for each cell in the RTM. Recall that each RCG node corresponds to exactly one method. If both values match, then the cell is expected to be correct. Otherwise, it will be tagged as a possibly erroneous cell. In both cases, we also supply the category through which the success or error was detected because we will see below that the different categories come with different likelihoods. The `getExpected()` algorithm distinguishes between inner nodes and leaf nodes. Its various if statements explore the surroundedness and border patterns with the dominance reflected in the ordering of the if statements. For example, category 1 is checked after category 2 because *trace dominates no trace*.

We investigated each category for correctness. Table 5.2 depicts the different categories with their respective coverage and correctness likelihoods in percentage. The coverage represents the ratio of cells satisfying the criteria of the category in question (e.g.,

		Category						
		1	2	3	4	5	6	7
Chess	Coverage	19.2	15.5	3.3	2.9	28.9	23.4	6.9
	Correctness	99.2	98.4	74.4	91.6	96.0	93.9	fail
Gantt	Coverage	35.8	2.6	5.8	3.2	38.2	6.5	7.9
	Correctness	98.6	92.7	81.7	77.2	96.5	67.3	fail
jHotDraw	Coverage	56.9	1.0	6.5	2.2	28.0	1.6	3.8
	Correctness	99.1	92.3	84.8	66.2	98.0	66.6	fail
VoD	Coverage	33.1	3.8	4.4	1.2	43.9	7.9	5.7
	Correctness	97.2	88.1	85.5	62.9	92.7	83.9	fail
Average Correctness		98.5	92.9	81.6	74.5	95.8	77.9	fail

Table 5.2: Coverage and correctness of categories in %

category 1 being n-surrounded only). The correctness refers to the ratio of correctly estimated traces links during the validation task. This data is again based on the golden standards of the four cases study systems. We see, for example, that the correctness of category 1 is very high (between 97-99%) and that this category applies to about a third of all RTM cells (19-56% of all cells). Other categories are quite good also, especially categories 2 and 5. Good but clearly inferior are the qualities for categories 3 and 4. This is expected since we discussed above that these categories mix border patterns with surrounding patterns which makes these cells more suspicious. However, it is important to note that these categories are quite rare: typically only 3-6% of all cells are of category 3 and 1-3% of all cells of category 4. The approach fails in a very low percentage of cells as indicated in category 7 (4-8% of cells cannot be validated).

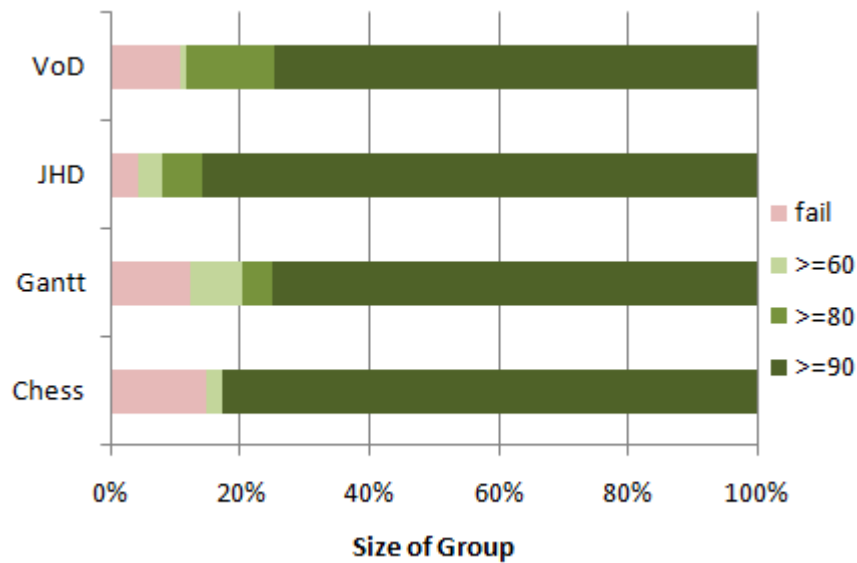


Figure 5.3: Distribution of correctness over coverage.

Figure 5.3 shows this distribution of traces validation quality relatively to the ratios of cells covered in the subject RTM. About 76-90% of cells were validated with a correctness higher than 90% (most of them $> 95\%$). These are mostly reached by the Categories 1, 2 and 5. The small remaining area is divided between cells of lesser validation quality and the small portion of cells that cannot be validated at all (leftmost area). It is important to note that our approach is able to validate most cells with high quality. The categories of failure and the cells with lower quality validations are few. It is the user's choice which categories to check manually.

Chapter 6

Tool Support: TraZer

Our approach is fully automatic and requires very specific input artifacts, namely a requirement trace matrix (RTM) and a call graph. We did implement this approach in a tool called *TraZer* which is basically an eclipse plug-in. In this chapter, we will introduce *TraZer* in details - especially the user interface and main functionalities of the tool. But before getting into the details of *TraZer*, It is important to understand the design decisions that we took before and during the implementation stage.

6.1 Design Decisions

Novel techniques, such as trace validation, are always tending to evolve very quickly. In this regard, it is important to create a generic implementation that eases later modifications and extensions. At the same time, we should consider the performance that we might lose due to the to a generic implementation. In this section, we will explain how did we succeed to implement *TraZer* on a generic platform, namely eclipse, and at the same keep our tool working with a high performance.

6.1.1 Eclipse as Platform

Our tool -*TraZer*- is implemented as a single eclipse plug-in. Eclipse ¹⁰ is, obviously, very known in the software development world. It is an open source integrated development environment (IDE) which is maintained by a not-for-profit corporation called Eclipse Foundation. The IDE provides a very generic plug-in platform that has been used for many years by other open source projects as well as commercial products. A

¹⁰<http://eclipse.org/>

wide variety of projects are already available on that platform without any licensing problems. Almost all projects are using one of the open source licenses, such as: Eclipse Public License (EPL), GNU General Public License (GPL), and Mozilla Public License (MPL)...

Our decision for eclipse is due to many reasons:

- Eclipse provides a lot of libraries that we could make use of. Indeed, the entire UI components of *TraZer* are implemented using JFace ¹¹, a UI toolkit that allows handling of many UI programming tasks. We also used the Zest ¹² toolkit for Visualization components, such as: drawing call graphs...
- There are already many other projects based in eclipse that we could later integrate with our current tool. So, using eclipse now would keep our implementation open for later integration with other projects.
- Eclipse is platform independent. The interface to the operating system is implemented in a separate layer and it will be always provided and maintained by the eclipse foundation. Thus, we do not need to spend additional effort to get our tool running on various platforms, such as: Windows X86, Windows X64, Linux, and MacOS ...
- There is also a personal motivation for using eclipse. After multiple years of professional working on eclipse, we have best “know how” of Eclipse Platform that should, on the one side, ease the implementation, and on the other one, improve the quality of the tool.

Eclipse provides also other tools which we might need during a validation task. It is certainly very useful to have all the tools integrated into one IDE. A software analyst or a developer would need to perform multiple operations on the source, e.g. browsing, searching for text, or even generating models out of it. We expect the trace validation task to be more convenient when the developer could use the same tool for other tasks that he/she also needs. On the one side, the developer would save the time of switching between one tool to another, and on the other side, he/she would save the effort of getting familiar with multiple tool at a time. Furthermore, as we mentioned before, eclipse is very known in software engineering and it is indeed widely used in commercial products. Most of developers and software analysts are already familiar with eclipse and used to work with it on daily basis. So, the learning effort required for our tool should be minor.

¹¹<http://wiki.eclipse.org/index.php/JFace>

¹²<http://www.eclipse.org/gef/zest/>

Our use of eclipse did not stop at the implementation level. The Eclipse Test and Performance Tools Platform ¹³ (TPTP) served to record the execution information of our case study systems. TPTP is a framework that provides multiple testing and profiling tools for open source as well as commercial products. More precisely, we used their debugging agent to record the calling relationships of methods at runtime. The recorded information is written into an external log file which could be later loaded into the TPTP allowing many other operations. For example, TPTP provides a very sophisticated UI for visualizing calling relationships (sequence diagram) and other profiling information (number of classes and instances ...). So far, we did not need any special integration with our tool. Both, our *TraZer* and TPTP, coexist independently in the eclipse platform. Installing TPTP is optional because our tool could also parse the execution files and acquire the needed information from them without any external libraries. Although TPTP provides a parser for the recorded files, we implemented our own parser in order to improve the overall performance of our tool.

Although eclipse is implemented with high quality standards, it still have few drawbacks. The plug-in based architecture has, unfortunately, a memory overhead due to the number of plug-ins that have to be loaded and managed at runtime. We did few experiments with a standalone version of our tool, but the saved memory consumption was not very convincing. Our tool has to deal with a huge amount of data (Execution Information, and RTM) at the same time. The memory overhead of the eclipse version compared to our standalone prototype did not exceed 100 Mb. Considering the advantages of eclipse, we did, then, take the final decision to implement *TraZer* as an eclipse plug-in.

6.1.2 Database

In most cases, our tool is supposed to deal with a huge amount of data. The recorded execution information is usually stored in files with several hundreds of Megabytes. Therefore, we had to carefully design the architecture of *TraZer*, so that the parsing time of files and memory consumption during the analysis should be acceptable. At the same time, the UI of our tool should be responsive even during the parsing and validation tasks. In that regard, eclipse provides very sophisticated design patterns called jobs and operations. They are similar to threads in conventional Java program but they will be created and managed more easily within eclipse. Therefore, we implemented all time consuming tasks in form of operations. They will run showing a progress information box which should inform the user about the tasks being done and their progress.

¹³<http://www.eclipse.org/tptp/>

The parsed information is kept in memory. We could have created an external database but the data needed is too huge and will be any way loaded to the cache of the database. Therefore, we decided to keep the parsed data into a custom virtual database which is implemented as a *Singleton* in order to prevent recreating it. Each virtual database is identified by the unique file name from which it was originally parsed. In the prototype implementation, we noticed that Java copies the virtual database to the objects requiring it in order to improve the performance of fetching data. This case occurs mostly when the database is defined as private field of another object. Unfortunately, this Java optimization affected drastically the memory consumption of the tool and produced an inconsistent state of the database in few cases. We were not able to reproduce the scenarios responsible for inconsistency states but took measures preventing it from happening again. The idea was simply to call the database by reference instead of using a private variable. Java does not support this operation natively. But there is a work around by using an intermediate referencing object. So, we defined a lightweight class called `DatabaseReference` having a single fields `databaseID` and a single method `getDB()` that fetches the concrete database from a `DatabaseFactory`. Figure 18 shows a class diagram modeling the database referencing mechanism in *TraZer*.

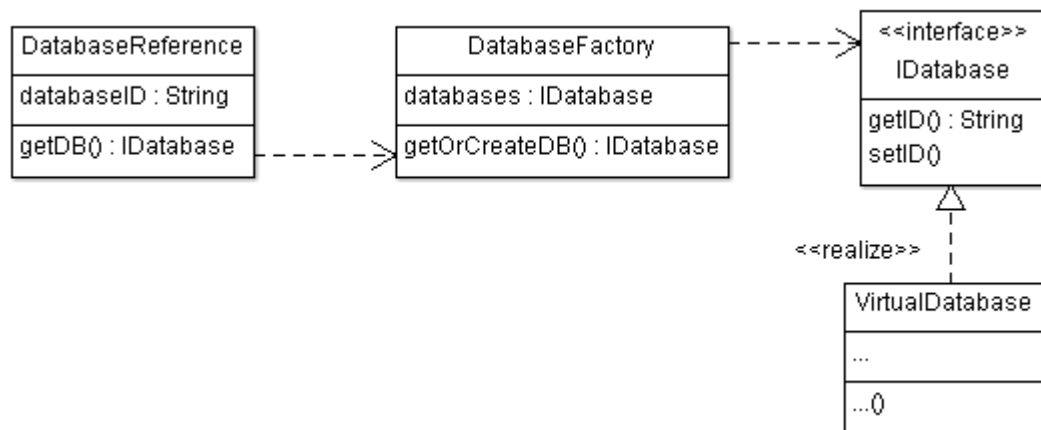


Figure 6.1: DatabaseReference class diagram.

The implementations of `IDatabase` are protected in the containing package and could only be created using the `DatabaseFactory`. Each database is created as a singleton which is uniquely identified by a `databaseID`. On the first call of `getOrCreateDB()` in the `DatabaseFactory`, a database will be initialized. For the moment, the `databaseID` is the project name for which the database has been created. The id is not limited to a specific pattern and we could use any possible `String`. If another database exists already with the given `databaseID`, it would be replaced with a new one. The other parts of the program should guarantee that each `databaseID` will be uniquely used. Thus, we could manage multiple databases at the same time with minimal memory consumption.

The objects requiring database access would use the lightweight `DatabaseReference` to call it using the `getDB()` method.

We are basically using a Builder design pattern. The `IDatabase` interface is the only visible data structure. `DatabaseFactory` will create databases using custom properties and it is the only object that could access class implementations of `IDatabase`, such as: `VirtualDatabase` which is the only implementation for the moment. This architecture would allow us to change the database implementation in later stages. For example, we could implement an SQL database that stores all the data in an external database instead of keeping it in the memory of our tool. This might be useful when the input data would exceed the memory limits of the hosting computer. We kept this possibility for later extensions and did not implement it during this thesis. All case study systems were analyzed with a memory consumption less than 1.5 Gb. We assume that in a nowadays standard computer would have enough memory ($> 4\text{Gb}$) to run our tool.

The huge amount of data is coming -in the first place- from the recorded runtime information of the subject projects. TPTP records such information in a standard XML format. A recorded file could very easily get to a size of multiple hundreds of Megabytes. For example, the execution information of `GanttProject` is about 310 Mb big. Parsing this data is presenting a challenge in the face of memory saving and usability constraints. There are multiple standard libraries for XML parsing in Java. Each of which have different characteristics and application areas. The most known of them are JDOM¹⁴ and SAX¹⁵. For example, JDOM creates an entire model of an XML file in the memory in order to let a quick and permanent access to all elements. This is useful when the XML document is defining elements which are referencing each other but introduced at different locations in the file. SAX, on the other side, does not keep anything in the memory. It does just parse the document elements and return them to the calling program which is supposed to take care of the element references consistency. The SAX library is, thus, much faster and requires less memory than the DOM library. We tested both libraries and found out that DOM library took in some case almost three times the size of the parsed file in memory and twice the parsing time of the SAX library.

At last, we decided to use the SAX library for parsing the TPTP recorded files. It proved to be very fast and we succeeded getting the 300 Mb files parsed in less than 10 seconds. The time consumption is very important in any UI based software because tasks with much time consumption would retard the responsiveness to the user even in multi-threaded systems. As mentioned above, we used eclipse operations to realize such “time-consuming” tasks in order to improve the responsiveness of our UI. Next, we present a detailed overview of the UI components that our tool -*TraZer*- provides.

¹⁴<http://www.jdom.org/>

¹⁵<http://www.saxproject.org/>

6.2 User Interface

Eclipse Platform has a very structured UI model. The components inside the main window are managed by another level of containers called perspective. Switching the perspective is meant to ease changing the entire content of the main window on the fly. We could define a perspective as a generic user interface that could be adapted to every tool individually. For example, in eclipse there is already a perspective for Java development, another for debugging, and another for Plug-ins development. . . This UI model allows to integrate a variety of tools into the same UI without getting confused between them. Usually, the perspectives share the same key-shortcuts and menu actions in order to keep a standard handling among the different tools.

Our tool implements also its own perspective containing UI components that should support the user during a trace validation task. Next, we will explain the structure of the *TraZer* perspective and how the provided components are meant to help the user.

6.2.1 Perspective

An eclipse perspective has a very clear structure. There is basically an editor-area, around which all other components are organized into sections depending on their location relatively to it. Figure 6.2 shows a screenshot of the *TraZer* perspective which constitutes of three main sections: (1) the project explorer view, (2) the editor-area, and (3) the utility views. Eclipse Platform provide the option to change the layout and the size of views individually. Each user could personally change the layout of the *TraZer* perspective and store it as a custom perspective layout. He/she could also extend the perspective by adding other sections and views from other tools. Basically, this option is provided to improve the integrity of tools with each other.

The *project explorer view* (1) is a tree view of projects being analyzed by the tool. Each project is created manually using a “wizard” that asks the user for the recorded execution log file and the requirement trace matrix (RTM) file. The first file is supposed to be an XML file in the standard TPTP format and the second one is an Excel file where the requirements are ordered on the columns side and the code elements on the row side. After creating a project, an automatic build should be launched in order to parse both files and create the appropriate artifacts (database, call graph file, and RTM file) that will be needed for the validation.

After creating a new project, the build will be started automatically. But existing projects have to be built manually when needed. There is a context menu action

that allows the user to call the *build project operation*. Although, Eclipse Platform provides the required techniques to automate the build operation, we did prefer keeping it managed manually, because once the user has multiple projects with big size input files, building all of them would need too much memory and -at a certain point- a usual computer won't be able to manage all of them at the same time.

Every file generated after building the project has an associated editor to view and -in some cases- also edit its content. The *editor-area* section (2) is the location where editors are opened when a file is selected in the project explorer view (1). The bottom section (3) contains *utility views* which is supposed to hold addition information views. In the course of this thesis, we implemented only one view from scratch, namely the "Patterns View". But we extended existing Eclipse views (Properties View and Console View) to show specific information related to the opened editor. In the remaining sections of this chapter, we describe the editors and views in the standard *TraZer* perspective with more details.

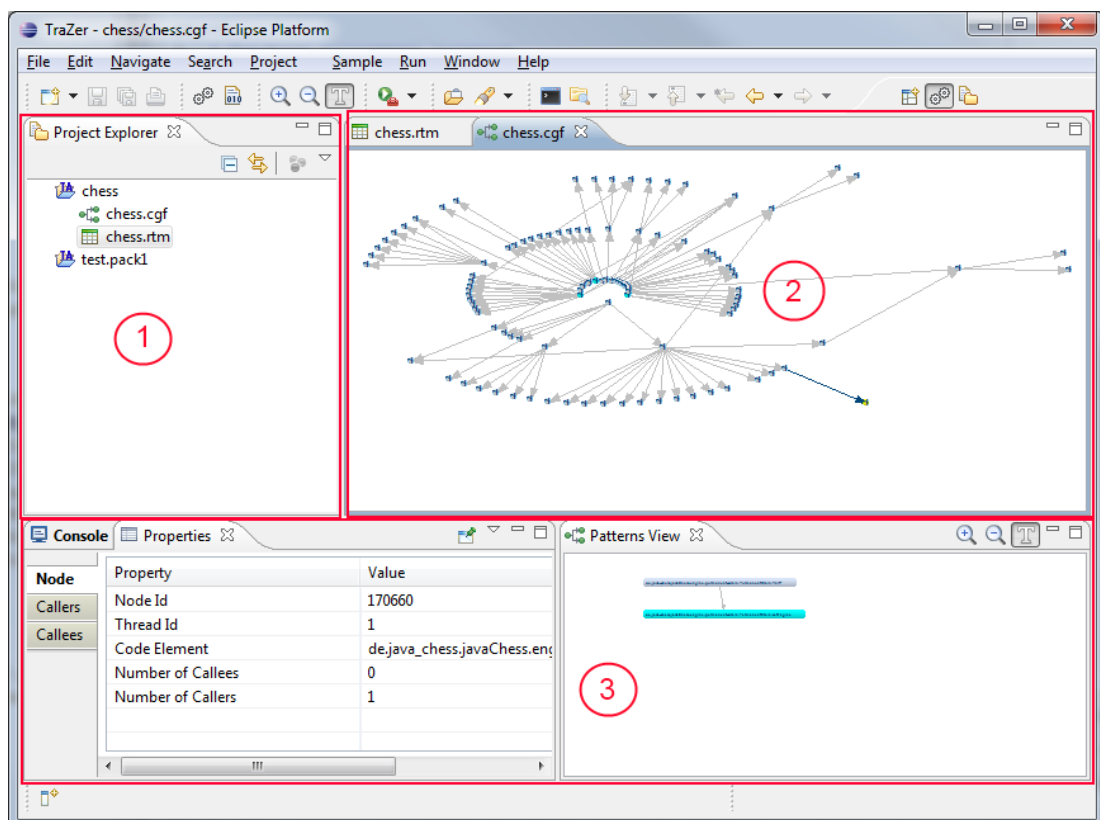


Figure 6.2: The *TraZer* perspective.




6.2.2 Editors

The editors are a conventional interface for viewing and editing file contents. In the Eclipse Platform, they are associated with unique features based on the multi-tools platform. For example, different editors could bind common actions from the tool bar and file menu to their content. This concept is very useful because it allows reusing existing actions for the same tasks even in different editors. The Eclipse platform grants the management of the opened files and their corresponding editors. It is responsible for the associations of editors and actions to each other. For each opened file the appropriate editor is initialized in a “tab” structure independently from all other editors. When an editor has the focus in the UI, the platform associates automatically the tool bar and file menu actions to it.

TraZer implements two editors which are intended to open the files produced during the project build. These files are just references for showing and editing the project data which is basically in the memory. We have mainly two files types: call graph files having the extension “.cgf” and requirement trace matrix files having the extension “.rtm”. Both of them do not have any content. Opening one of the files would just call the appropriate editor and load the data that it requires directly from the database of the project. Thus, the editors would not show any content if the appropriate project is not built. We recall that the build operation will be called automatically after creating a new project, but the user has to call it manually for existing projects.

Call Graph Editor

The call graph editor is the editor responsible for opening the “.cgf” files. It shows the call graph as it was generated from the execution log file which has been previously defined when creating the project. We use the Zest (see section 6.1.1 on page 58) library to create a graphical representation of the call graph. It creates a graphical node for each method and connects the nodes to each other using the calling relationships of the methods. Each node is labeled with the name of the code element which it is representing. The result would be a visible directed graph indicating the calling relationships between code elements. In section (2) of figure 6.2, you can see an example content of the call graph editor.




The editor contributes to the toolbar with few useful actions for graphical operations, such as: Magnify Nodes () , Shrink Nodes () , and Show/Hide Node Labels (). These actions are supposed to provide simple tools to manipulate the graph in order to improve the visibility of the graph. Some projects have hundreds of nodes and Zest will reduce the size of nodes in order to fit them all into the available space. The Mag-

nify/Shrink Nodes actions provide the option to increase or reduce the size of nodes. This is especially very helpful when the nodes are getting very small. Furthermore, the user might hide or show the labels of the nodes in order to make a better overview picture of the call graph without being disturbed by the text appearing on each node.

RTM Editor

There is a massive problem with RTM representations. Using Excel sheets has been the common sense representation of RTMs, even though there is no standard representation for RTMs yet. Using a common table manipulation software, such as: Excel, might do the needed tasks, but usually, they are commercial and do not provide easy interfaces for external access to the content of files. Every time we find an open source library for reading and writing Excel files, we face other unexpected problems. For example, the Apache POI ¹⁶ (the Java API for Microsoft Documents) deceived us with its huge memory consumption especially when writing files. Therefore, we decided not to use external tools for RTMs and build our own table editor that provides the required functions for an RTM.

The screenshot of the RTM editor depicted in figure 6.3 shows an excerpt of the Chess RTM. As explained previously, the table maps the code elements (rows) to the requirements (columns). We use our naming convention for traces 'T' and no-traces 'N' in every cell to define the type of the link between the given code element and requirement. We are also using colors for the different links: green for 'T' and orange for 'N'. Thus, we keep consistent names and symbols between RTMs of the different projects independently from their format and symbols.

The table in RTM editor contains three levels of information: the Input level, the Estimation level, and the Validation level. The user could switch between the different levels using the available actions in the top right corner of the editor. Each of them has its own action icon representing the initials of the level name (Input: ; Estimation: ; Validation: ). By default, the input level is shown when the RTM editor is opened. It shows basically the input RTM as it was read from the input file. The estimation level reveals the estimated values of our algorithm. Additionally to the terms and colors of the input level, the estimation level depicts cells that could not be validated as failed 'F' in light-yellow color. The validation level exhibits the comparison between previous levels. If the estimated value confirms the input value, then the cell would be marked as correct with the green color. Otherwise, it will be marked as a possible erroneous link in a red color. But, as we have multiple degrees of certainty about the correctness (recall the categories 1 to 7), we are applying a different color

¹⁶<http://poi.apache.org/>

Method \ Requirement	R0	R1	R2	R3	R4
de.java_chess.javaChess.GameController.-init-(Lde/java_...	T	N	N	T	N
de.java_chess.javaChess.GameController.computerPly()Z	T	N	N	T	N
de.java_chess.javaChess.GameController.convertUserPly(...	T	N	N	T	N
de.java_chess.javaChess.GameController.doPly(Lde/java...	T	N	N	T	N
de.java_chess.javaChess.GameController.gameOver(Z)Z	T	N	N	T	N
de.java_chess.javaChess.GameController.getBoard()Lde/j...	T	N	N	T	N
de.java_chess.javaChess.GameController.getEngine()Lde...	T	N	N	T	N
de.java_chess.javaChess.GameController.getGame()Lde/j...	T	N	N	T	N
de.java_chess.javaChess.GameController.getGameNotati...	T	N	N	T	N
de.java_chess.javaChess.GameController.getGameTimer(...	T	N	N	T	N
de.java_chess.javaChess.GameController.getRenderer()L...	T	N	N	T	N
de.java_chess.javaChess.GameController.reset()V	T	N	N	T	N
de.java_chess.javaChess.GameController.setBoard(Lde/ja...	T	N	N	T	N
de.java_chess.javaChess.GameController.setEngine(Lde/j...	T	N	N	T	N
de.java_chess.javaChess.GameController.setGame(Lde/ja...	T	N	N	T	N
de.java_chess.javaChess.GameController.setGameNotati...	T	N	N	T	N

Figure 6.3: The RTM Editor.

tone for each category. Presumably correct links of category 1 and 2 are said to have the best chances for correctness, and thus we mark them with a dark green color. We note, here, that both traces and no-traces are handled equally and marked with dark green color that indicates a correct link. The category 3 and 4 are weaker but still have a considerable chance of correctness. Therefore, we color them in lighter green indicating possible locations for manual checks. For incorrect links, on the other hand, we are using different tones of red color. Again, the darker tone is marking the categories 1 and 2, and the lighter tone is marking the categories 3 and 4. The cells where our approach failed to compute an estimation are kept unmarked.

6.2.3 Views

In the Eclipse Platform a view is typically a small section showing particular data or information. For example, the Project Explorer View shows the tree of projects, their

sub-files, and sub-folders in the workspace. The Eclipse Platform is already providing a lot of views. Any new tool could implement its own views from the scratch or modify and reuse existing ones. In the *TraZer* project we created three views using three different strategies.

Console View

The console view is already provided by Eclipse Platform. We applied a mechanism to extend it into a custom view. The standard view in eclipse is supposed to show the output of java programs. Our extended version is called *TraZer Console*. When *TraZer* starts, this console hooks itself to the Log4J ¹⁷ library in order to track the logging output from our tool. It does also provide a more friendly user interface than the standard console by coloring the output lines depending on the severity of the log statement: error logs are displayed in red, warnings in orange, and info and debug statements are displayed respectively in blue and green. Thus, it is much easier to recognize a failure in the program by just tracking the red color statements.

Properties View

The Properties View is also already provided by Eclipse Platform. There is however no need to extend the existing one. It is already implemented in a generic way. The Properties View reacts basically to each selection in the workbench and shows the appropriate properties of the selected data when possible. The selected objects must provide an interface to let the properties view access its properties and show them. Each object, which is supposed to accept such access, must implemented an adapter (design pattern) from the type `IPropertySource`. We implemented this design pattern for the graph nodes and the RTM cells. When one of these objects (a node in the graph or a cell in the RTM editor) is selected, the Properties View checks whether it is extending the `IAdapter` interface and providing an adapter for the `IPropertySource` type or not. If it is the case, the view calls the adapter to get the properties of the selected object.

Patterns View

The Patterns View is actually the only view that we created from scratch. Following the concept of the Properties View, we designed it also to listen to selections in the

¹⁷<http://logging.apache.org/log4j/>

RTM Editor and Call Graph Editor. It shows basically a detailed view of the call graph starting at the selected method. If we select node in the Call Graph Editor, we show a sub graph containing that node and all its callers and callees. On the one hand, the advantage of this view over the Call Graph Editor is that it shows the connections more clearly when the Call Graph is overcrowded with too many nodes. On the other hand, if we select a cell in the RTM, the view shows the node of the corresponding code element and its callers and callees in the call graph without the need to switch back to the call graph editor. Additionally, in this view the nodes are colored as their cells do in the RTM. Such a view should help the user understand the decisions taken by our tool by manually checking the calling and called code elements and their labels.

Chapter 7

Evaluation

Our approach was designed to validate existing traces. It is thus independent of any technique used to capture or recover the trace links. In this section, we demonstrated that our approach is able to validate trace links with high correctness and coverage where correctness reflects the percentage of correct error feedback and coverage the percentage of RTM cells that can be validated. Ideally, both correctness and coverage should be high. Since the golden standards represent high-quality trace information but trace validation is as likely used on less perfect trace information, this section demonstrates our approach's ability to cope with erroneous traces where we show that correctness remain high and coverage drops to more incorrect the traces.

7.1 Accuracy

Thus far, we demonstrated that our approach has both high accuracy and high coverage in context of the four case study systems and their respective golden standards. Our approach's correctness should, however, be high also for RTMs of less perfect quality (for which the quality is not known *a priori*). Next, we measure the performance of our approach with varying input quality.

Errors are basically wrong 't's (a 't' which is supposed to be an 'n') and/or wrong 'n's ('n' which is supposed to be 't') in the RTM. We thus randomly seeded errors in the golden standards by switching 't's into 'n's and 'n's into 't's to measure our approach's ability to detect incorrect traces and incorrect no traces. We tested our approach on RTMs with 10-50% random trace errors seeded. The percentage was measured relative to the number of traces in the RTM. That is, if a requirement was implemented in twice as many methods than another requirement then 10% seeding of errors involved twice as many methods.

The traces are usually much less than no-traces, especially in larger projects. We noticed that in the case study systems the number traces for each requirement was about 5-12%. The Chess system was the only one with a balanced number of traces and no-traces. For this reason the error injections was not as intuitive as it might seem to be. There is a dilemma about the number of errors that we are supposed to inject in order to make a meaningful comparison between traces and no-traces. For example, injecting 10% of erroneous traces (turning 10% of 'n's into 't's) might produce twice the number of traces that do really exist in a big project, and thus the number of wrong traces would be about 50% instead of 10% as we intuitively thought. Therefore we decided to scale the number of injected errors relatively to the number of traces in the given project. The number of random 'n's seeded was matched the number of random 't's seeded to allow for direct comparison. We injected both random 't's and random 'n's but measured them separately to understand their respective implications.

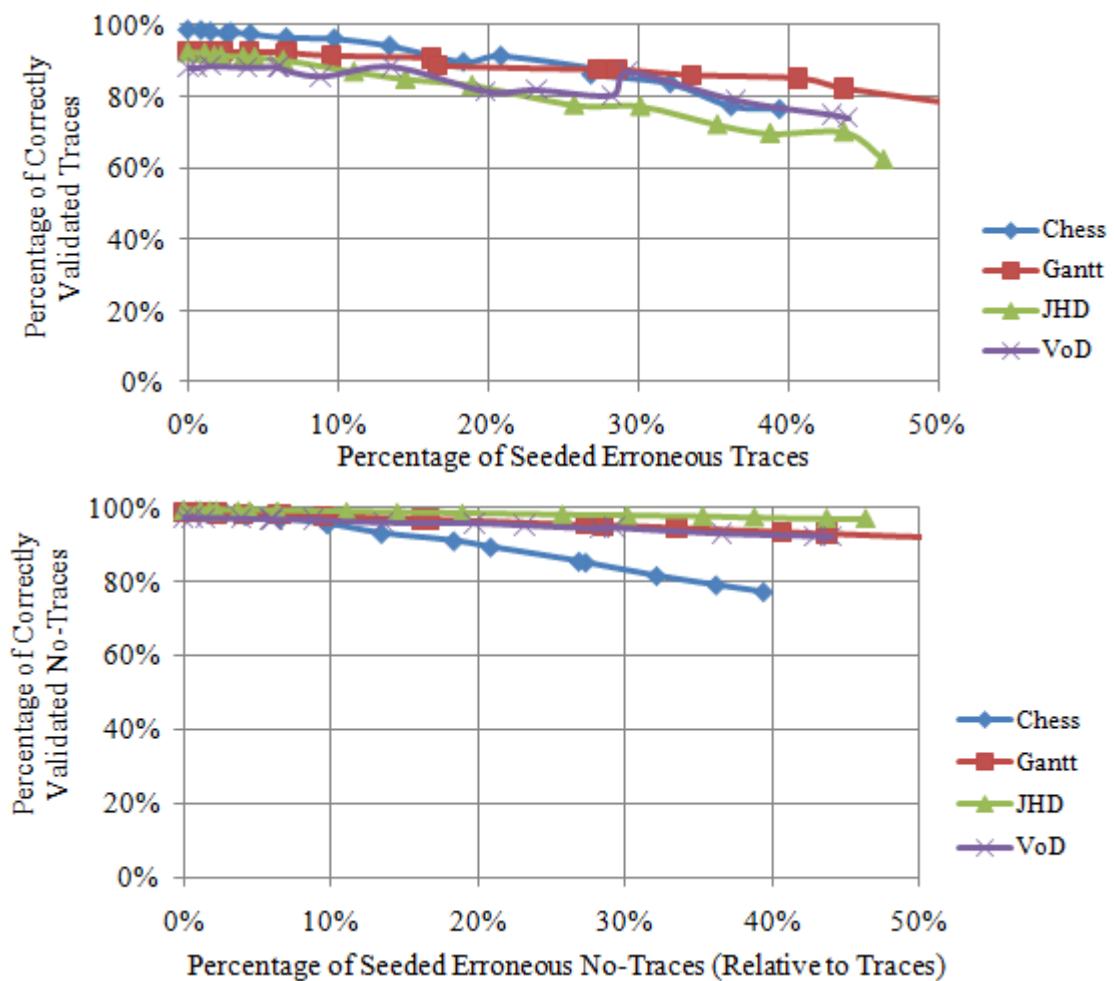


Figure 7.1: Validation quality of traces/no-traces in poor quality RTMs.

We applied our algorithm on such RTMs seeded with errors and recorded the percentage of correct validations. Figure 7.1 shows the evolution of correct trace and no-traces ratio with increasing errors ratio. We observed that our approach performs quite well with $< 30\%$ seeded errors. Both traces and no-traces were well above 80% correctly validated. The upper part in Figure 7.1 shows data with regard to the random seeding in traces categories 2 and 4 only. No-trace categories 1 and 3 shows an even better behavior with the same test RTMs used for trace links (lower part in Figure 7.1). Except for the Chess system, all other projects had almost a constant correctness ratio even at very high errors ratios. The correctness decreased between $0-3\%$ from the initial values. This could be explained by the difference between the initial number of no-trace links and the number of seeded errors. The Chess project did not have a big difference between the number of traces and no-traces, and thus, the correctness ratio of no-traces decreased similarly to that of traces.

The seeded errors in a test RTM could have three possible effects on the RCG structure:

1. Weaken existing surrounding cases by adding boundary patterns to surrounded methods.
2. Break existing surroundings and change them to boundaries.
3. Create new surroundings.

The first effect (1) would affect methods with multiple pure surroundings. If one of those surroundings has been changed, our algorithm would still recognize the method in question as a possible link. It would just shift to a less correctness certainty category but the estimate link would still correctly generated. Although, we would risk losing good quality links, this effect does not influence the coverage of neither traces nor no-trace.

The second effect (2) is very dangerous because it would directly affect the best quality category of trace/no-trace links. If a method has only one pure surrounding, it will lose its surrounding by only changing one of the neighboring links. In this case, the surrounding pattern will be transformed into a boundary pattern and the method won't be correctly estimated any more but rather eliminated by the Filtering Rule. It is more dangerous for the coverage than for the quality of the results.

The last effect (3) is the most dangerous one, since it would create incorrect surroundings. We distinguish two kinds of this effect: (i) is when a boundary patterns is transformed into a surrounding, and (ii) is when both sides of a surrounding are

coincidentally changed and thus, the opposite surrounding would be created. The kind (i) is not very problematic. The boundary pattern identifies already a method that has the same chances having a trace or a no-trace link. Intuitively, random errors would introduce as much incorrect links as correct ones. This kind should not present a big threat to the quality. It might, however, increase the coverage by introducing links that were originally eliminated by the Filtering Rule. But the coverage improvement should stay small, since the ratio of boundary patterns is very limited ($< 5\%$ in most test projects). The second kind (ii) influences the accuracy of best quality links category that we were able to classify by pure surroundings. We expect this kind of errors to affect both correctness as well as coverage of our approach. But, as the chances are very small getting both erroneous neighbors, the influence on the overall quality of the results should be still acceptable.

7.2 Coverage

The coverage in the context of this work is the metric defining the ratio of links that could be validated from the total number of links. We did previously (figure 5.3 on page 56) show that our approach was covering above 80% of trace links. In this section we aim at investigating the impact of decreasing RTM quality on the coverage of validated traces.

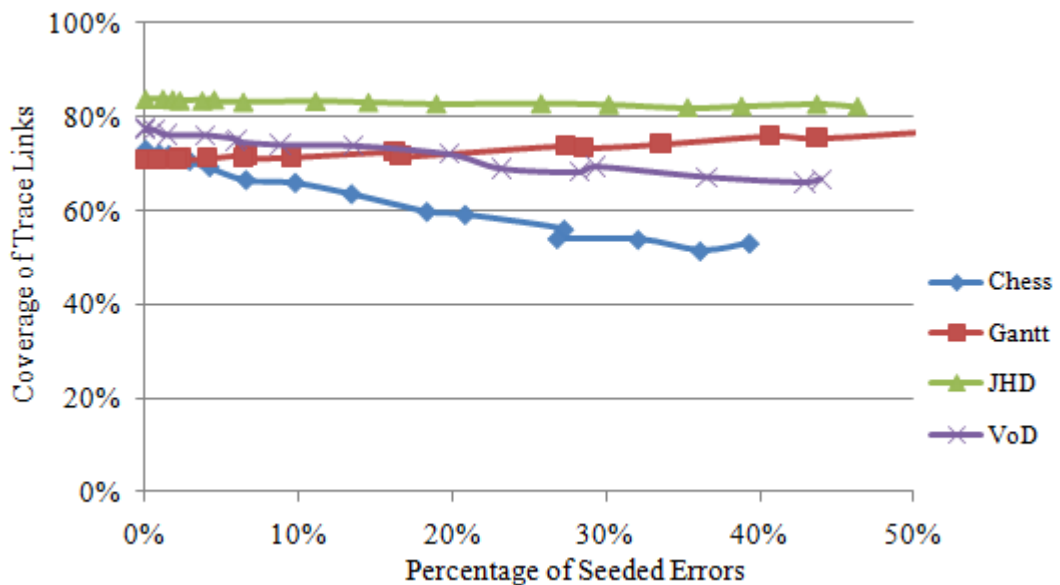


Figure 7.2: Validation coverage for traces.

Figure 7.2 depicts the coverage of our approach relative to the random error seeding applied in previous section. Generally, we see that the coverage also decreases slightly with increasing error seeding. The effect is especially visible in the Chess system which is a very special case study. We recall that Chess has a roughly balanced number of traces and no-traces. Consequently the number of seeded error was higher especially for no-traces. In fact, the high number of wrong traces and no-traces is the main reason for the remarkable loss of correctness and coverage. The other systems had almost a constant coverage over all test RTMs.

Overall, the correctness and coverage metrics confirm our approach's applicability to validating requirements to code traces of various degrees of quality. Although, the case study systems have different characteristics, the performance of our approach seems to be very reliable even in some special cases (Chess system). There is also another property that we should exploit, namely scalability. An approach that does not scale with growing size of input might get very quickly to its limits and be useless for many commercial products.

7.3 Scalability

The main goal of our approach is to automate the validation of traces because this task is as time consuming as traces retrieval. The tool support provided by TraZer should reduce the overall effort required. But, usually there is a problem with automating growing problems, namely scalability. It is important to analyze how our algorithm scales with bigger projects. Unfortunately, we did not have the chance to get bigger projects than the actual case study systems. Therefore we limit the scalability analysis to these systems.

The algorithm that we are proposing in this thesis (see section 5.3 on page 54) is very simple and straightforward. Considering the two-dimensional structure of the RTM, the asymptotic complexity of that algorithm is $O(n \times m)$, where n is the number code elements and m is the number of requirements. There is also the possibility to implement the matrix as simple array with a sophisticated index computation as used to be in hash-tables. Such an implementation might improve the performance of the algorithm but the asymptotic complexity would remain the same. An experimental analysis of the performance of our tool on the case study systems showed that our tool requires less than 10 seconds for the biggest projects (JHotDraw and GanttProject). But most of the time is required for parsing the input data. The validation is always finished in few fractions of a second which could be explained by the linear complexity of the algorithm.

Chapter 8

Discussion and Future Work

There is already a lot of activity in the area of traceability but the automation of retrieval and recovery of traces has always been the first priority task. We think that the automation is less useful as long as the correctness of traces is not validated. A reliable validation of retrieved traces would certainly improve their quality. Our approach is already validating traces with very good correctness ratios and we believe that it is a very good candidate for combinations with other retrieval and recovery approaches. There are, however, some open issues that we have to investigate before getting into analyzing the combinations of our approach with other automated traceability approaches. In this chapter, we discuss shortly some of the open issues that have to be investigated first before building other techniques upon it.

In this work, we met implicitly some assumptions which are not fulfilled in some cases. E.g. we assumed that the golden standard RTMs are complete and correct for the case study systems. But in the course of our work, we noted that the completeness is not always fulfilled. We might argue that the assumption should be correct, since the quality delivered by our approach is satisfactory. But we should prove that those open issues do not represent any threat to validity of our approach. Therefore, an extensive investigation is necessarily to show the effect of such issues and prove that their effect is limited and would not threaten the reached results. We discuss in following sections some of these issues and propose our expectations about their effects.

8.1 Incompleteness

We have always assumed that the input data is complete and correct. The correctness should be already guaranteed by the professional developers responsible for retrieving the traces. Practically speaking, there is no reasonable method to get a traces quality

better than what such a developer could deliver. Therefore, we find our assumption about the correctness very sound and we admit it to be true. But the question of completeness is still open nonetheless. Since we have two types of input data, we distinguish between two kinds of incompleteness: On the one side, the input RTM could be incomplete, and on the other one, the call graph could be missing some calls.

The incompleteness of an RTM means missing traces/no traces for some code elements. We distinguish between two cases: full incompleteness and partial incompleteness. The first case of full incompleteness occurs when a code element is missing or its entire row is empty in the RTM. The second case occurs when only few cells are empty or missing, and hence the incompleteness is partially. We faced the full incompleteness case in our study multiple times. We recall that in our study only a sample set of requirements treated was analyzed in each system. There are other requirements which have not been traced, and consequently the code elements tracing to them would be missing in the RTM. The second case of missing cells is mainly due to human mistakes. The developer recovering the golden standard RTM could have overseen some cells. We should remember that there is a “tiredness” (see section 2.4 on page 32) effect which would influence the performance of a human during the recovery or validation of traces.

The cells of the RTM are basically used as labels in the requirement call graphs (RCG). When some of them are missing, we would get some missing labels in the RCGs. For this thesis, we have decided to keep the incompleteness issue out of the scope. So, we did simply ignore the nodes with missing labels in RCG. Our algorithm seemed to be delivering very good results even without any special treatment for missing traces. Therefore, we expect such kind of missing data to have a very limited influence on the overall quality of the validation algorithm that we are proposing.

The incompleteness in the call graph is generally caused by the recorded execution logs. Some calling relationships could have been missed due to poorly selected unit tests. Thus, selecting the unit tests carefully is of high importance for our approach. In our work, it is not try to prove that we performed good or bad unit tests. Our main goal was creating a simple approach without any high effort preparations. Therefore, we did not take any special care about the quality of the unit tests. We have just executed each requirement as it is supposed to be done by a normal user. In spite of that, the quality and performance of our approach are very convincing.

The study of missing calls’ effect might be the subject of a future work. For the moment, our approach did not show any quality degradation due to missing calls or missing traces. We suggest also to run our approach on a static call graph. In fact, a static call graph might provide the missing calls from the dynamic call graph. Even

though, we have intentionally chosen the dynamic analysis to eliminate the overhead of dead code, we might need a static analysis to fill the gaps resulting by poorly designed unit tests.

8.2 Granularity

The granularity issue could be discussed on both sides of a trace, on the code element side as well as on the requirements side. The granularity deals basically with the level of details in trace links. Traces that are connecting methods to requirements are certainly different from traces connecting classes to features (high level requirement).

On the code element side, the granularity defines the details level to which the traces are connected. E.g. are the traces connected to methods, to classes, or to lines of code... we assumed in the scope of this thesis that the traces are on methods level. Unfortunately, there are no known standards about the granularity of code elements in traceability. All case studies we used are providing traces from requirements to classes. The method level traces are easily derived from the class level. We recall a basic concept in traceability that *some code is said to trace to a given requirement if this code implements the requirement in part or full* (see section 5.1 on page 51). Therefore, we derive the traces of methods from their classes. Each method in a class will be linked to the requirements that the enclosing class is tracing to in the RTM. Unfortunately, the derivation introduces some erroneous traces especially in classes which are implementing multiple requirements at the same time.

The patterns concept that we introduced in this thesis is based on the idea of having requirements implemented in multiple methods which would call each other at runtime. This concept might be useless if the requirements are implemented in a higher (e.g. classes) or lower (e.g. lines of code) granularity. We did not investigate this but we think such an investigation could make our assumptions more objective, especially that tracing is required in different levels of granularity. Furthermore, we expect the information of lower granularity relationships to introduce new patterns. For example, the methods and fields in a class could be connected with new patterns. The study of such relationships and patterns would be certainly enriching for our approach. As a consequence, the algorithm and the quality categories have to be extended in light of the new patterns and relationships.

On the requirement side, there is also an issue about the requirements details level. As we have already explained in a previous section (see section 2.1 on page 7), there is a difference between a software feature and a software requirement. Many developers do

not know the difference between them. Usually, the feature is a high level requirement which is comprehensive to a client. In some cases, it is also called a concern, because it is concerning -in the first place- the people who are supposed to use the final product. A feature/concern does often summarize a set of low level requirements. For our investigation, the requirements in the golden standard RTMs might be classified as features for two reasons: (1) in many cases, individual requirements are grouping too many methods in their call chains which is an indication for big requirements (features); (2) the investigated requirement call graphs have often had more than one call chain. Each call chain might represent an independent low level requirement, but they are grouped under the same feature.

Further investigations on requirements/features composition could be useful for our approach as well as for others. We imagine that understanding the call chains structure could be very helpful for program understanding and feature location researches. Unfortunately, we were not able to achieve this step in the scope of this thesis due to the lack of permanent work with the responsible developers on the case study systems.

8.3 Traces Prediction

Although, this thesis is mainly proposing an automated technique for requirement-to-code traces validation, other purposes could be served using the same technique. Our approach does automatically estimate trace link for methods using other existing trace links and then compares it to the actual trace link in order to estimate the correctness of that trace. But our approach is also able to estimate a trace link even when the actual value is missing (incompleteness). Obviously, it is not a matter of validation anymore but rather a prediction of the missing trace link.

We have shown that our approach is delivering high quality estimations. In case of missing traces, the developer might apply our approach to predict possible links for methods with sufficient neighboring traces (surroundedness). The delivered traces are ordered into several quality categories and the developer could decide which traces or categories are suspicious and have to be verified manually. For example, the developer might decide to keep all traces acquired from full-surrounding cases (correctness $> 90\%$) and verify the other categories. The verification of predicted traces would be certainly less complicated and less time consuming than the retrieval from scratch. Moreover, the traces prediction could be called recursively and generate further traces based on predicted ones. The quality of recursively generated traces might deteriorate compared to the traces which are predicted in the first run. The topic of trace prediction is certainly worth further investigation, especially that it is mixing the validation with the coverage of traces.

There are some behaviors that we did not mention in the observations chapter, because they are of no relevance to our validation approach. We noted, for example, that the call chains are often very near to each other so that only few methods are separating them. In our opinion, the separating methods are very strong candidates for erroneous no-traces. So, they should be connecting the call chains as traces rather than separating them as no-traces. This kind of locations (between two call chains) in a requirement call graph might be expressed in other patterns longer than three elements. Then, we would be able to exactly identify the methods involved in such separations of call chains. Yet, another possible extension for our approach.

8.4 Conclusion

This thesis demonstrated a novel approach to validating requirements-to-code traces together with calling relationships in the code. We introduced the algorithm of our approach which first computes expected traces/no-traces based on the surroundedness patterns and then compares them with the given input traces. If the given traces differ from the expected trace then an error is reported. The error is a potential error since our heuristics are not 100% correct but, as was demonstrated, our approach is of high quality and applicable to most cells.

Our technique represents an important step towards more automation for traceability because state-of-the-art predominantly focused on creating or recovering requirement-to-code trace links where the correctness of these retrieved traces are typically verified manually. Yet, manual trace validation was shown to be of poor quality with experiments showing that human subjects are likely decreasing the quality of input traces that exceeded $> 50\%$ correctness [18]. Our approach is able to validate $> 80\%$ of the input traces with $> 90\%$ correctness. Only 4-8% of the traces could not be validated. We also demonstrated that the approach is still very reliable even with relatively poor quality input.

We admit that there are still other open issues worth investigating, but the proposed technique has, nonetheless, proved the ability to provide reliable traces validation with a high correctness. We have also shown that the same technique could be used for other purposes, such as: traces prediction. Obviously, this thesis is opening new opportunities in the traceability area, but at the same time, it is also rising other research questions about the correlation between traces and execution information.

Bibliography

- [1] Antoniol, Canfora, Casazza, and De Lucia. Information retrieval models for recovering traceability links between code and documentation. In *Proceedings International Conference on Software Maintenance ICSM-94*, pages 40–49, Victoria, BC, Canada, 2000.
- [2] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo. Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering*, 28(10):970–983, October 2002.
- [3] G. Antoniol, G. Canfora, A. De Lucia, and E. Merlo. Recovering code to documentation links in OO systems. In *Sixth Working Conference on Reverse Engineering (Cat. No. PR00303)*, pages 136–144, Atlanta, GA, USA, 2002.
- [4] G. Antoniol and Y. -G Gueheneuc. Feature identification: a novel approach and a case study. In *Proceedings of the 21st IEEE International Conference on Software Maintenance, 2005. ICSM'05*, pages 357– 366. IEEE, September 2005.
- [5] Ted J Biggerstaff, Bharat G Mitbander, and Dallas Webster. The concept assignment problem in program understanding. In *Proceedings of the 15th international conference on Software Engineering, ICSE '93*, pages 482–498, Baltimore, Maryland, United States, 1993. IEEE Computer Society Press. ACM ID: 257679.
- [6] Kunrong Chen and Václav Rajlich. Case study of feature location using dependence graph, after 10 years. In *2010 IEEE 18th International Conference on Program Comprehension*, pages 1–3, Braga, Portugal, June 2010.
- [7] J. Cleland-Huang, C.K. Chang, and M. Christensen. Event-based traceability for managing evolutionary change. *IEEE Transactions on Software Engineering*, 29(9):796–810, September 2003.
- [8] Jane Cleland-Huang, Brian Berenbach, Stephen Clark, Raffaella Settimi, and Eli Romanova. Best practices for automated traceability. *Computer*, 40(6):27–35,

June 2007.

- [9] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke. A systematic survey of program comprehension through dynamic analysis. *IEEE Transactions on Software Engineering*, 35(5):684–702, September 2009.
- [10] Marc Eaddy, Alfred V. Aho, Giuliano Antoniol, and Yann-Gaël Guéhéneuc. CERBERUS: tracing requirements to source code using information retrieval, dynamic analysis, and program analysis. In *2008 The 16th IEEE International Conference on Program Comprehension*, pages 53–62, Amsterdam, The Netherlands, June 2008.
- [11] Alexander Egyed, Florian Graf, and Paul Grunbacher. Effort and quality of recovering Requirements-to-Code traces: Two exploratory experiments. In *2010 18th IEEE International Requirements Engineering Conference*, pages 221–230, Sydney, Australia, September 2010.
- [12] Andrew David Eisenberg and Kris De Volder. Dynamic feature traces: Finding features in unfamiliar code. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 337–346. IEEE Computer Society, 2005. ACM ID: 1091864.
- [13] O.C.Z. Gotel and C.W. Finkelstein. An analysis of the requirements traceability problem. In *Proceedings of IEEE International Conference on Requirements Engineering*, pages 94–101, Colorado Springs, CO, USA, 1994.
- [14] Jane Huffman Hayes, Alex Dekhtyar, and James Osborne. Improving requirements tracing via information retrieval. In *Proceedings of the 11th IEEE International Conference on Requirements Engineering*, page 138, Washington, DC, USA, 2003. IEEE Computer Society. ACM ID: 943920.
- [15] J.H. Hayes, A. Dekhtyar, and S.K. Sundaram. Advancing candidate link generation for requirements tracing: the study of methods. *IEEE Transactions on Software Engineering*, 32(1):4–19, January 2006.
- [16] J.H. Hayes, A. Dekhtyar, S.K. Sundaram, and S. Howard. Helping analysts trace requirements: an objective look. In *Proceedings. 12th IEEE International Requirements Engineering Conference, 2004.*, pages 233–243, Kyoto, Japan, 2004.
- [17] Michael Jiang, Michael Groble, Sharon Simmons, Dennis Edwards, and Norman Wilde. Software feature understanding in an industrial setting. In *2006 22nd IEEE*

- International Conference on Software Maintenance*, pages 66–67, Philadelphia, PA, USA, September 2006.
- [18] W. Kong, J.H. Hayes, A. Dekhtyar, and J. Holden. How do we trace requirements? an initial study of analyst behavior in trace validation tasks. In *Fourth International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE 2011)*, May 2011.
- [19] Patrick Mäder, Orlena Gotel, and Ilka Philippow. Rule-Based maintenance of Post-Requirements traceability relations. In *2008 16th IEEE International Requirements Engineering Conference*, pages 23–32, Barcelona, Spain, September 2008.
- [20] 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 '03*, pages 125–135, Washington, DC, USA, 2003. IEEE Computer Society.
- [21] Marius Marin, Arie Van Deursen, and Leon Moonen. Identifying crosscutting concerns using Fan-In analysis. *ACM Transactions on Software Engineering and Methodology*, 17(1):1–37, December 2007.
- [22] Collin McMillan, Denys Poshyvanyk, and Meghan Revelle. Combining textual and structural analysis of software artifacts for traceability link recovery. In *2009 ICSE Workshop on Traceability in Emerging Forms of Software Engineering*, pages 41–48, Vancouver, BC, Canada, May 2009.
- [23] Tao Qin, Lu Zhang, Zhiying Zhou, Dan Hao, and Jiasu Sun. Discovering use cases from source code using the branch-reserving call graph. In *Software Engineering Conference, 2003. Tenth Asia-Pacific*, pages 60– 67. IEEE, December 2003.
- [24] Martin P. Robillard and Gail C. Murphy. Concern graphs: finding and describing concerns using structural program dependencies. In *Proceedings of the 24th international conference on Software engineering - ICSE '02*, page 406, Orlando, Florida, 2002.
- [25] M.P. Robillard, W. Coelho, and G.C. Murphy. How effective developers investigate source code: an exploratory study. *IEEE Transactions on Software Engineering*, 30(12):889–903, December 2004.
- [26] Norman Wilde and Michael C. Scully. Software reconnaissance: Mapping program features to code. *Journal of Software Maintenance: Research and Practice*,

7(1):49–62, January 1995.

- [27] Stefan Winkler and Jens Pilgrim. A survey of traceability in requirements engineering and model-driven development. *Software & Systems Modeling*, 9(4):529–565, December 2009.
- [28] W.E. Wong, S.S. Gokhale, J.R. Horgan, and K.S. Trivedi. Locating program features using execution slices. In *Proceedings 1999 IEEE Symposium on Application-Specific Systems and Software Engineering and Technology. ASSET'99 (Cat. No. PR00122)*, pages 194–203, Richardson, TX, USA, 1999.
- [29] Wei Zhao, Lu Zhang, Yin Liu, Jiasu Sun, and Fuqing Yang. SNIAFL: towards a static noninteractive approach to feature location. *ACM Transactions on Software Engineering and Methodology*, 15(2):195–226, April 2006.

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benützt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Linz, am July 20, 2011

Achraf Ghabi