

UNIVERSITY OF CALIFORNIA, SAN DIEGO

Mining Test Cases To Improve Software Maintenance

A dissertation submitted in partial satisfaction of the
requirements for the degree of Doctor of Philosophy

in

Computer Science

by

Celal Ziftci

Committee in charge:

Professor Ingolf Heiko Krüger, Chair
Professor William G. Griswold
Professor William S. Hodgkiss
Professor Ryan Kastner
Professor Kevin M. Patrick

2013

Copyright
Celal Ziftci, 2013
All rights reserved.

The Dissertation of Celal Ziftci is approved and is acceptable in quality
and form for publication on microfilm and electronically:

Chair

University of California, San Diego

2013

DEDICATION

To God

... for giving me what it takes to make this contribution and be happy in life.

To my mother

... for loving me more than anything in the world,
and giving up her life so that I could have one.

To my father

... who wanted me to be a 'Doctor', but could not live to see this day.

To my family

... for letting me pursue my dreams for years so far away from home.

EPIGRAPH

The true sign of intelligence is not knowledge but imagination.

Albert Einstein

Program testing can be used to show the presence of bugs,
but never to show their absence.

Edsger Dijkstra

Only a life lived for others is a life worthwhile.

Albert Einstein

TABLE OF CONTENTS

Signature Page	iii
Dedication	iv
Epigraph	v
Table of Contents	vi
List of Figures	x
List of Tables	xiv
Acknowledgements	xvi
List of Abbreviations	xviii
Vita	xx
Abstract of the Dissertation	xxii
Chapter 1 Introduction	1
1.1 Software Development Processes	2
1.2 Software Maintenance: the Most Costly Activity in the Software Development Lifecycle	8
1.3 Understanding Requirements is Important	9
1.4 Software Testing and Using Tests to Improve Maintenance	10
1.5 Limitations of Previous Work	12
1.6 Main Contributions and Dissertation Outline	18
Chapter 2 Running Example and Background	21
2.1 Running Example	21
2.2 Background	22
2.2.1 Program Analysis Techniques	22
2.2.2 Requirements	23
2.2.3 Features	26
2.2.4 Requirements and Features: Connecting the Dots	30
2.2.5 Requirements Tracing	33
2.2.6 Feature Location	34
2.3 Case Studies	35
2.3.1 Finding Requirements	38
2.3.2 Creating Scenarios	38
2.3.3 Collecting Execution Traces	38

2.4	Commonly Used Evaluation Metrics	39
2.4.1	Precision, Recall, F-Measure	39
2.5	Classification of the Work In This Dissertation	41
Chapter 3	Feature Location Using Data Mining on Existing Test Cases	43
3.1	Introduction	43
3.2	Related Work	46
3.3	Limitations of Existing Techniques	52
3.4	Contribution	54
3.5	FLMINER: Feature Location Miner	57
3.5.1	Input to FLMINER	58
3.5.2	Association Rule Learning and Confidence	59
3.5.3	Divergence: Detecting Edge Cases	63
3.5.4	Affinity: Combining Confidence and Divergence	64
3.5.5	Outputs: Feature Locations, Affinity and User Guidance	65
3.6	Evaluation	66
3.6.1	Baseline for Comparison: the Base Case	66
3.6.2	Ideal Case	68
3.6.3	Practical Case	73
3.7	Discussion	80
3.7.1	Threats to Validity	81
3.8	Conclusion	82
3.9	Future Work	83
Chapter 4	Tracing Requirements to Tests with High Precision and Recall	84
4.1	Introduction	84
4.2	Related Work	87
4.3	Contribution	92
4.4	FORTA: Tracing Requirements to Tests Via Features	93
4.4.1	Features and Scenarios	95
4.4.2	Feature Markers	95
4.4.3	Tracing Features to Test Cases	97
4.4.4	Tool Support	98
4.5	Evaluation	98
4.5.1	Input to FORTA	99
4.5.2	Inputs to TF-IDF and LSI	100
4.5.3	Results for Requirements Traceability Links	100
4.6	Discussion	100
4.6.1	Threats to Validity	105
4.7	Conclusion	106
4.8	Future Work	108

Chapter 5	Test Intents: Enhancing the Semantics of Requirements Traceability	
	Links in Test Cases	109
5.1	Introduction	109
5.2	Background	110
5.3	Contribution	112
5.4	Finding Test Intents	112
5.4.1	Formal Concept Analysis	114
5.4.2	Finding Test Intents Using FCA	116
5.4.3	Tool Support	117
5.5	Evaluation	118
5.5.1	Finding the Baseline for Test Intents	118
5.6	Discussion	119
5.6.1	Threats to Validity	120
5.7	Related Work	121
5.8	Conclusion	121
5.9	Future Work	122
Chapter 6	Automatically Mining Requirements Relationships From Test Cases	123
6.1	Introduction	123
6.2	Related Work	127
6.3	Contribution	129
6.4	REQRELEX: Mining Requirements Relationships from Test Cases	130
6.4.1	Mining Requirements Relationships	132
6.4.2	Minimizing the Number of Requirements Dependencies	133
6.4.3	Tool Support	134
6.5	Evaluation	135
6.5.1	Input to REQRELEX	136
6.5.2	Inputs to Other Approaches.....	136
6.5.3	Evaluation Criteria	136
6.6	Discussion	136
6.6.1	Threats to Validity	144
6.7	Conclusion	145
6.8	Future Work	147
Chapter 7	Requirements Testing Progress: How Well Does the Test Suite of a System Cover Its Requirements?	148
7.1	Introduction	148
7.2	Contribution	152
7.3	Related Work	153
7.4	REQTESTPRO: Requirements Level View for Testing Progress	156
7.4.1	Step 1: Inputs —RTM and Requirement Priorities.....	156
7.4.2	Step 2: Find Weighted-RTM	158
7.4.3	Step 3: Calculating Progress for Each Requirement.....	161

7.4.4	Step 4: Output	163
7.5	Evaluation	164
7.5.1	Case Studies	164
7.5.2	Evaluation Criteria	165
7.5.3	Evaluation Results	166
7.6	Discussion	172
7.6.1	Threats to Validity	176
7.7	Conclusion	176
7.8	Future Work	177
Chapter 8	Conclusion and Future Outlook	180
8.1	Conclusion	180
8.2	Future Outlook	184
8.2.1	Mapping Requirements and Features	184
8.2.2	Addressing Non-Functional Requirements	185
8.2.3	Considering the Valuation of Tests	185
8.2.4	Integration Into Production Development Environments	186
	Bibliography	187

LIST OF FIGURES

Figure 1.1.	Steps in the Waterfall Model [127], adapted from [127]. The development moves from one phase to the other in one direction. If requirements change, the process starts from the beginning.	3
Figure 1.2.	Steps in Model Driven Development [11]. The development moves from one phase to the other in one direction. If requirements change, the process starts from the beginning. Different from the Waterfall Model [127], the system code and tests are automatically or semi-automatically generated from the formal requirements specification.	3
Figure 1.3.	A sample user story in Behavior Driven Development [114]. A story can have multiple test scenarios that describe the expected inputs, actions and system behavior. Scenarios are defined using 'Given' to describe the expected starting conditions and/or system state, 'When' to describe an action and/or input, and 'Then' to describe the expected output or system state.	4
Figure 1.4.	Steps in Test Driven Development [76, 23, 89]. Requirements are gathered via user stories first. Then tests are implemented to capture them. Afterwards, the production system is implemented until all tests pass. Once an iteration is complete, the system is expanded with new user stories in upcoming iterations until the whole system is developed.	5
Figure 1.5.	Phases of the Spiral Model, courtesy of Boehm [31]. The development moves from one phase to the other as in the Waterfall Model [127]. Iterations provide increments that expand the scope of the system while reducing risk.	7
Figure 1.6.	Convection cycle between requirements and tests.	12
Figure 1.7.	High level summary of the existing techniques to form a convection cycle between requirements and tests. * signs indicate where the work in this dissertation falls.	13
Figure 1.8.	Recording relationships between requirements themselves and with tests manually.	14
Figure 1.9.	Getting relationships between requirements themselves and with tests automatically or semi-automatically using MDD [11].	15

Figure 1.10.	Mining relationships between requirements themselves and with tests by mining them from the existing tests.	16
Figure 1.11.	Outline of the dissertation. Numbers in each box are the chapter numbers where each work is explained in detail.	19
Figure 2.1.	The relationship between requirements and features. In our Chat System example, features that exist in the system map one-to-one to the requirements of the system, while there are also non-functional requirements of the system, which are not captured by the features as we define them here.	28
Figure 2.2.	Sample execution unit for Java [68]. The initial number is the call depth, while the rest is a concatenation of the class name and the method name, along with the location of the method in the source code.	29
Figure 2.3.	Sample execution trace of the send-message feature of the Chat System. The caller - callee relationships are reflected in the execution trace with tabs.	31
Figure 2.4.	The overall flow of our activities to analyze our case studies.	37
Figure 2.5.	Sets describing precision, recall and f-measure.	40
Figure 3.1.	Source code location suggestions for each feature in the Chat System. Feature location suggestions are ranked from more to less relevant. A developer can start investigating the implementation of a feature starting from these feature location suggestions.	44
Figure 3.2.	A simple code snippet that can calculate the area of an isosceles triangle, an equilateral triangle or a rectangle (example from Wong et al. [152]).	47
Figure 3.3.	The lines triggered by each scenario: (a) The original code snippet shown in Figure 3.2 (from Wong et al. [152]), (b) The lines triggered when scenario t_1 is executed, (c) The lines triggered when scenario t_2 is executed, (d) The lines triggered when scenario t_3 is executed.	48
Figure 3.4.	Automation of feature scenarios. Scenarios in Table 2.2 are automated as small programs.	53

Figure 3.5.	Overview of FLMINER, with inputs, its process and outputs. The inputs are the test-cases labeled with the features they execute. The outputs are the highly relevant feature location suggestions and feedback on the quality of the suggestions.	57
Figure 3.6.	Sample input to FLMINER for the Chat System.	58
Figure 3.7.	FLMINER performance with $2n$ fuzzy test cases as input (n is the number of features), compared to the base case input [119].	79
Figure 4.1.	Tracing requirements in tests. We start with the descriptions of requirements (the requirements specification document), and the aim is to get to the requirements traces in tests, in the form of an RTM in our case.	86
Figure 4.2.	Precision and recall in the context of tracing requirements in tests. The relevant set is the set of requirements traces (the ticks) in the RTM between requirements and tests. The retrieved set is the set of requirements traces suggested by a technique. Precision, then, is the correctness of the requirements traces suggested by the technique, while recall is the completeness of the actual requirements traces with respect to the suggested traces by the technique.	88
Figure 4.3.	Classification of related work for FORTA.	89
Figure 4.4.	Inputs, outputs and flow of FORTA. The inputs are execution traces of scenarios and tests, while the output is the Requirements Traceability Matrix between requirements and tests. The steps with a * indicate a novel contribution of our approach, while steps with a + indicate that we use existing research and make a contribution additionally.	94
Figure 5.1.	The steps in finding test intents.	113
Figure 5.2.	Concept lattice for the formal context shown in Table 5.4.	116
Figure 6.1.	Dependencies between requirements of the Chat System.	124
Figure 6.2.	Hint-relations between some of the requirements of Apache Pool [3].	125
Figure 6.3.	Inputs, outputs and steps of REQRELEX. Execution traces of the scenarios and test cases are the inputs, while the requirements dependencies and hint-relations are the outputs.	131

Figure 6.4.	Sample results on minimizing the number of dependencies found in the Chat System via transitive reduction. This graph is a reduced version of the one shown in Figure 6.1.....	134
Figure 7.1.	The number of test cases implemented by the developers for each requirement in Apache Pool [3]. Each bar in the chart corresponds to a requirement, and the height of the bar shows how many tests were implemented for a requirement.	150
Figure 7.2.	The number of bugs reported for each requirement in Apache Pool [3]. Each bar in the chart corresponds to a requirement, and the height of the bar shows how many bugs were reported for a requirement.	151
Figure 7.3.	Inputs, steps and outputs of REQTESTPRO. The inputs are the RTM and requirement priorities. The output is the testing progress view over requirements.	157
Figure 7.4.	Concept lattice for the formal context shown in Table 7.3.	160
Figure 7.5.	Testing Progress visualized as a bar chart.	164
Figure 7.6.	The testing progress of Apache Pool that REQTESTPRO outputs along with the number of bugs reported for each requirement in Apache Pool [3]. Many requirements that are considered to have inadequate testing by REQTESTPRO (the ones under the line at the top chart) have a higher number of bugs reported.	173
Figure 8.1.	Overview of the contribution of this dissertation. We provided a holistic approach to using tests as a useful source of information on requirements and we developed an end-to-end process to benefit from the testing phase during the development and maintenance of a system.	183

LIST OF TABLES

Table 2.1.	Requirements specification document for our running example Chat System. Each requirement is assigned a unique identifier, a name and a description.	25
Table 2.2.	Features and scenarios for each feature for the Chat System	26
Table 2.3.	Sample Requirements Traceability Matrix for tests of the Chat System	34
Table 2.4.	Properties of the case studies used in this dissertation.	36
Table 2.5.	Classification of the work in this dissertation in different dimensions.	42
Table 3.1.	Classification of related work on feature location based on two dimensions.	44
Table 3.2.	The items and database based on execution traces for the test cases in Figure 3.6.	61
Table 3.3.	Divergence values for each feature and affinity values for each (method, feature) pair. Highlighted cells (in bold and blue color) show which methods would be chosen as feature markers for each feature.	66
Table 3.4.	Information about case studies used to evaluate FLMINER	73
Table 3.5.	Average f-measure based on the number of labeled test cases provided as input for each case study in the evaluation of FLMINER. .	80
Table 4.1.	FORTA case study properties.	99
Table 4.2.	Requirements traceability results.	101
Table 4.3.	Scenario creation methodologies compared on Apache Commons CLI [1].	101
Table 5.1.	Sample Requirements Traceability Matrix for Tests of the Chat System (duplicate of Table 2.3 for convenience).	111
Table 5.2.	Sample Requirements Traceability Matrix for tests enhanced to show test intents for the Chat System.	111

Table 5.3.	Sample Requirements Traceability Matrix for tests enhanced to rank requirements according to likelihood to be test Intents for the Chat System.	111
Table 5.4.	Concepts in the formal context for the Chat System RTM given in Table 5.1	115
Table 5.5.	Case study properties to evaluate finding test intents.	118
Table 5.6.	Test intent mining results.	119
Table 6.1.	REQRELEX case study properties.	135
Table 6.2.	Case study results on finding requirements relationships	138
Table 6.3.	Comparison of the methods compared with REQRELEX	142
Table 7.1.	Priorities assigned to each requirement in the Chat System. A higher number corresponds to a higher priority, i.e. a more important requirement.	158
Table 7.2.	Sample Requirements Traceability Matrix for the tests of the Chat System (duplicate of Table 2.3 for convenience).	158
Table 7.3.	Concepts in the formal context for the Chat System RTM given in Table 7.2	160
Table 7.4.	Weighted-RTM for the RTM given in Table 7.2 (geometric sequence base = 10)	161
Table 7.5.	Notation used to calculate the progress scores in REQTESTPRO. ...	162
Table 7.6.	Properties of the case studies used in the evaluation of REQTESTPRO.	165
Table 7.7.	Percentage distances of progress scores between those found by REQTESTPRO and the ground truth*	167
Table 7.8.	Minimum, maximum and average percentage distances of progress scores between those found by REQTESTPRO and the ground truth in 100.000 regression runs.	169
Table 7.9.	Percentage distances of progress scores between those found by REQTESTPRO and the ground truth under varying geometric bases*.	171

ACKNOWLEDGEMENTS

The journey of a Ph.D. student is typically considered a solo endeavor. However, while taking this journey, you inevitably have or meet people who have made a profound impact in your life.

First and foremost, I would like to thank my mother, sister and family for their continuing and endless support and confidence during my PhD journey. My mother has been both a mother and a father for me after I lost my father at the age of seven. She has been making critical decisions that had a huge positive impact on all aspects of my life, she has been supporting me to live a life of dreams of mine and my father's, and she gave up her entire life just so I can live better. There is no way for me to fully show my appreciation to her, but I dedicate to her everything I have achieved in life, including my PhD.

Second, I was fortunate enough to meet several people who had a positive influence on me, and made my grad school experience enjoyable. I would like to thank my advisor, Professor Ingolf Krüger, for his overall encouragement and advice. I also thank Professors William G. Griswold and Kevin Patrick for their financial support and for their generosity in widening my view of the computing world. I thank all the members of my dissertation committee for reading this manuscript and providing feedback. I thank my friends in San Diego for their emotional support and encouragement, and for acting as my immediate family during the tough times and endless nights of my PhD. I thank Barry Demchak and Massimiliano Menarini for their advice during my PhD career. Finally, I thank Barry Demchak, Massimiliano Menarini, To-Ju Huang, Filippo Seracini, Xiang Zhang and Yan Yan for their friendship and support as my office mates.

Portions of this dissertation are based on papers that have been published in various conferences, several of which I have co-authored with others. Listed below are the details along with my contributions.

Chapter 3, in full, is a reprint of the material as it appears in: “Celal Ziftci and Ingolf Krüger. Feature Location Using Data Mining on Existing Test-Cases. In the 19th Working Conference on Reverse Engineering, pages 155-164, Kingston, Ontario, Canada, 2012. IEEE.” The dissertation author was the primary investigator and author of this paper.

Chapter 4, in full, is a reprint of the material as it appears in: “Celal Ziftci and Ingolf Krueger. Tracing Requirements to Tests With High Precision and Recall. In Proceedings of the 26th International Conference on Automated Software Engineering, pages 472-475, Lawrence, Kansas, USA, November 2011. IEEE.” The dissertation author was the primary investigator and author of this paper.

Chapter 5, in full, is a reprint of the material as it appears in: “Celal Ziftci and Ingolf Krüger. Test Intents: Enhancing the Semantics of Requirements Traceability Links in Test Cases. In Proceedings of the 28th ACM Symposium On Applied Computing, pages 1272-1277, Coimbra, Portugal, 2013. ACM.” The dissertation author was the primary investigator and author of this paper.

Chapter 6, in full, is currently being prepared for submission for publication of the material. The dissertation author was the primary investigator and author of this material.

Chapter 7, in full, is a reprint of the material as it appears in: “Celal Ziftci and Ingolf Krüger. Getting More From Requirements Traceability: Requirements Testing Progress. In Traceability in Emerging Forms of Software Engineering, pages 12-18, San Francisco, California, USA, 2013. IEEE.” The dissertation author was the primary investigator and author of this paper.

LIST OF ABBREVIATIONS

FLMINER	Feature Location Miner
FORTA	Feature Oriented Requirements Traceability Analysis
REQRELEX	Requirements Relationship Extractor
REQTESTPRO	Requirements Level View for Testing Progress
API	Application Programming Interface
BDD	Behavior Driven Development
COTS	Commercial Off-The Shelf
EQ	Evaluation Question
FCA	Formal Concept Analysis
GUI	Graphical User Interface
IR	Information Retrieval
LSI	Latent Semantic Indexing
MC/DC	Modified Condition/Decision Coverage
MDD	Model Driven Development
NIST	National Institute of Standards and Technology
OO	Object Oriented
RE	Requirements Engineering
RFC	Request For Comments

RT	Requirements Traceability/Requirements Tracing
RTM	Requirements Traceability Matrix
TDD	Test Driven Development
TF-IDF	Term Frequency Inverse Document Frequency
UCSD	University of California, San Diego
US	The United States of America

VITA

- 2004 Bachelor of Science
Department of Computer Science, Bilkent University, Turkey
- 2004–2006 Teaching Assistant
Department of Computer Science, University of Illinois, Urbana Champaign, USA
- 2006 Master of Science
Department of Computer Science, University of Illinois, Urbana Champaign, USA
- 2006–2009 Software Developer
Morgan Stanley, New York, USA
- 2009–2013 Research Assistant
Department of Computer Science, University of California, San Diego, USA
- 2013 May Doctor of Philosophy
Department of Computer Science, University of California, San Diego, USA
- 2013 Aug– Software Engineer
Google, New York, USA

PUBLICATIONS

Celal Ziftci and Ingolf Krueger, “Tracing Requirements to Tests With High Precision and Recall,” in Proceedings of the 26th International Conference on Automated Software Engineering, pp. 472-475, Lawrence, Kansas, USA, November 2011. IEEE.

Celal Ziftci and Ingolf Krüger, “Feature Location Using Data Mining on Existing Test-Cases,” in the 19th Working Conference on Reverse Engineering, pp. 155-164, Kingston, Ontario, Canada, October 2012. IEEE.

Celal Ziftci and Ingolf Krüger, “Test Intent: Enhancing the Semantics of Requirements Traceability Links in Test Cases,” in Proceedings of the 28th ACM Symposium On Applied Computing, pp. 1272-1277, Coimbra, Portugal, March 2013. ACM.

Celal Ziftci and Ingolf Krüger, “Getting More From Requirements Traceability: Requirements Testing Progress,” in the 7th International Conference on Traceability in Emerging

Forms of Software Engineering, pp. 12-18, San Francisco, California, USA, May 2013. IEEE.

Nima Nikzad, Celal Ziftci, Piero Zappi, Nichole Quick, Priti Aghera, Nakul Verma, Barry Demchak, Kevin Patrick, Hovav Shacham, Tajana S Rosing, Ingolf Krueger, William G Griswold, Sanjoy Dasgupta, "CitiSense - Adaptive Services for Community-Driven Behavioral and Environmental Monitoring to Induce Change," Technical report CS2011-0961, University of California San Diego, January 18 2011.

Elizabeth Bales, Nima Nikzad, Nichole Quick, Celal Ziftci, Kevin Patrick and William Griswold, "Citisense: Mobile Air Quality Sensing for Individuals and Communities," in the 6th International Conference on Pervasive Computing Technologies for Healthcare, pp. 155-158, San Diego, California, USA, May 2012. IEEE.

Celal Ziftci, Nima Nikzad, Nakul Verma, Piero Zappi, Elizabeth Bales, Ingolf Krueger and William Griswold, "Citisense: Mobile Air Quality Sensing for Individuals and Communities," in Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity, pp. 23-24, Tucson, Arizona, USA, October 2012. ACM.

Nima Nikzad, Nakul Verma, Celal Ziftci, Elizabeth Bales, Nichole Quick, Piero Zappi, Kevin Patrick, Sanjoy Dasgupta, Ingolf Krueger, Tajana Simunic Rosing, William G. Griswold, "CitiSense: Improving Geospatial Environmental Assessment of Air Quality Using a Wireless Personal Exposure Monitoring System," in Proceedings of the Conference on Wireless Health, article 11, San Diego, California, USA, October 2012. ACM.

Kevin Patrick, Laura Wolszon, Karen M Basen-Engquist, Wendy Demark-Wahnefried Alex V Prokhorov, Stephanie Barrera, Chaitan Baru, Emilia Farcas Ingolf Krueger, Doug Palmer, Fred Raab, Phil Rios, Celal Ziftci, Susan Peterson, "CYberinfrastructure for COMparative Effectiveness REsearch (CYCORE): Improving Data From Cancer Clinical Trials," in the Journal of Translational Behavioral Medicine, pp. 83-88, vol. 1, iss. 1, March 2011, Springer-Verlag.

ABSTRACT OF THE DISSERTATION

Mining Test Cases To Improve Software Maintenance

by

Celal Ziftci

Doctor of Philosophy in Computer Science

University of California, San Diego, 2013

Professor Ingolf Heiko Krüger, Chair

Software development comprises of several phases, including but not limited to requirements gathering, design, development, verification & validation, and maintenance. Software development processes are frameworks that impose structure on building a software system, using one or more of the phases above, and they are broadly classified as plan-driven and agile. Plan-driven processes (e.g. the Waterfall Model) follow a rigid structure on the order of phases from requirements gathering towards maintenance, in the order given above. They put emphasis on documentation, repeatability and stability of the phases. On the other hand, agile processes (e.g. Test Driven Development) follow an

iterative and incremental approach, where the phases can be repeated while the scope of the system is expanded on each iteration. Agile processes put more emphasis on system artifacts (source code and tests) than documentation, which makes them more suitable for the work in this dissertation.

Although software systems can be built following different development processes described above, maintenance is the dominating cost during the lifetime of a system, with 70%-90% of the total cost. During maintenance, the dominating activity is program comprehension, i.e. understanding requirements and their relation with the system artifacts such as source code and tests. Therefore, obtaining as much information about requirements as possible is a major concern during maintenance.

A common activity during the verification & validation phase of development is testing. It is reported that, in a typical software project, approximately 50% of the total development time is expended on testing. With the advent of agile processes, this number is even higher. During testing, test results are typically used in a binary fashion, i.e. to see if they pass or fail. However, tests contain more information about requirements that is useful to stakeholders during maintenance.

In this dissertation, we develop novel techniques to understand what is captured in tests and exploit this information to provide a better understanding on the relationships between requirements, and their relationship with tests. We provide a holistic approach to using tests as a useful source of information on requirements and we develop an end-to-end automated process to benefit from the testing phase during the development and maintenance of a system.

Chapter 1

Introduction

Software development comprises of several phases from the gathering of requirements to the deployment of the final product. Requirements analysis and definition phase typically consists of establishing the system's services, constraints and goals through consultation with the system's users. Requirements are then typically defined in detail and recorded to serve as a specification of the system [133]. In the system design phase, an overall architecture is established, and the fundamental system abstractions and their relationships are described [133]. In the development phase, the design of the system is realized as a set of programs [133] so that the system provides the services as identified in the requirements gathering phase. In the verification & validation phase, the system produced during the development phase is checked to make sure it satisfies and conforms to the requirements. Finally, in the maintenance phase, any and all changes that need to be made on the developed system are performed. The reasons for maintenance include: changing requirements and updating the system according to the updated requirements, adding new requirements, fixing bugs, and refactoring the system to improve its internal structure.

A software development process (also known as software development lifecycle) is a set of activities and the corresponding results which produce a software system [133]. Software development processes are frameworks that impose structure on developing

software systems. They propose going through several phases in succession or iteratively until the system to be developed is completed. Although each process is different in the outline it follows, all processes typically share several or all phases described above, albeit possibly in varying forms.

1.1 Software Development Processes

Software development processes are broadly categorized as plan-driven and agile [29]. These two categories of processes have different characteristics, and advantages and disadvantages.

Plan-driven processes are considered as the traditional way of building a system, where the development typically moves through the phases of requirements gathering, designing, building, verification & validation, deployment and maintenance. The Waterfall Model [127] is a well-known member of this family of processes. In this model, the process moves through the steps in one direction from gathering the requirements to the finished system, and puts emphasis on documentation and review at each step to allow verification of the activities performed (see Figure 1.1). The movement between stages is only allowed once a phase is completed and reviewed to confirm that it is indeed complete. Another plan-driven process is Model Driven Development (MDD) [11]. In this model, as shown in Figure 1.2, requirements are gathered, converted into a formal specification, and the software system along with its tests is then automatically or semi-automatically generated from the formal requirements specification through several formal transformations. Similar to the Waterfall Model, the system development typically moves in one direction. If requirements change, system development is continued by going to the first step. The advantage of the plan-driven processes is that they provide predictability, repeatability and stability throughout the development of the system. Since documentation is important in these processes, the stakeholders know where to look for

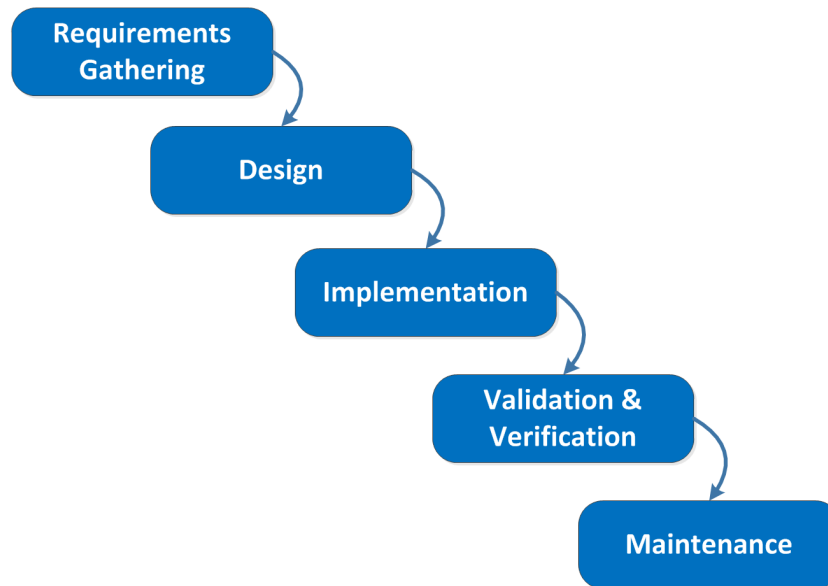


Figure 1.1. Steps in the Waterfall Model [127], adapted from [127]. The development moves from one phase to the other in one direction. If requirements change, the process starts from the beginning.

information. The disadvantages of these processes stem from their rigid structure on moving between the phases. The response to changing requirements is slow, since the phases must typically be followed through from the beginning. In addition, documenting all activities in each step can take a considerable effort, and put constraints on the speed of development [29].

Due to the disadvantages of the plan-driven processes, agile processes were pro-



Figure 1.2. Steps in Model Driven Development [11]. The development moves from one phase to the other in one direction. If requirements change, the process starts from the beginning. Different from the Waterfall Model [127], the system code and tests are automatically or semi-automatically generated from the formal requirements specification.

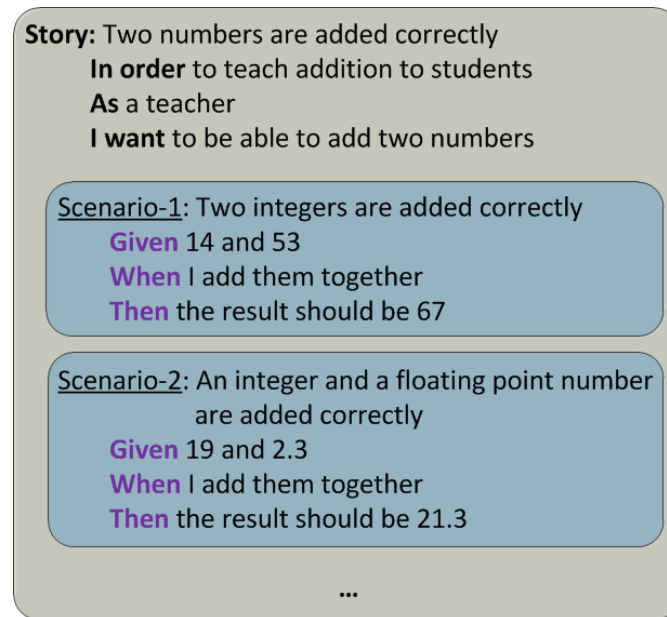


Figure 1.3. A sample user story in Behavior Driven Development [114]. A story can have multiple test scenarios that describe the expected inputs, actions and system behavior. Scenarios are defined using 'Given' to describe the expected starting conditions and/or system state, 'When' to describe an action and/or input, and 'Then' to describe the expected output or system state.

posed on the other end of the spectrum. Agile processes still perform the typical software lifecycle phases such as requirements gathering, design, development, verification & validation, deployment and maintenance. However, these steps are realized in short iterative cycles incrementally, and they actively involve stakeholders to put forth and verify requirements. They put emphasis on user stories and system artifacts to realize those stories, such as the system implementation and test cases. Examples of these processes are Test Driven Development (TDD) [76, 23, 89] and, an evolution of TDD, Behavior Driven Development (BDD)[114]. In these processes, first, test cases are implemented to capture requirements in the form of user stories (see Figure 1.3 for a sample user story captured in BDD). Then, the system is implemented to make sure the test cases pass and realize these user stories. The overall flow of agile processes is shown in Figure 1.4). The development continues incrementally, where at each iteration new user stories

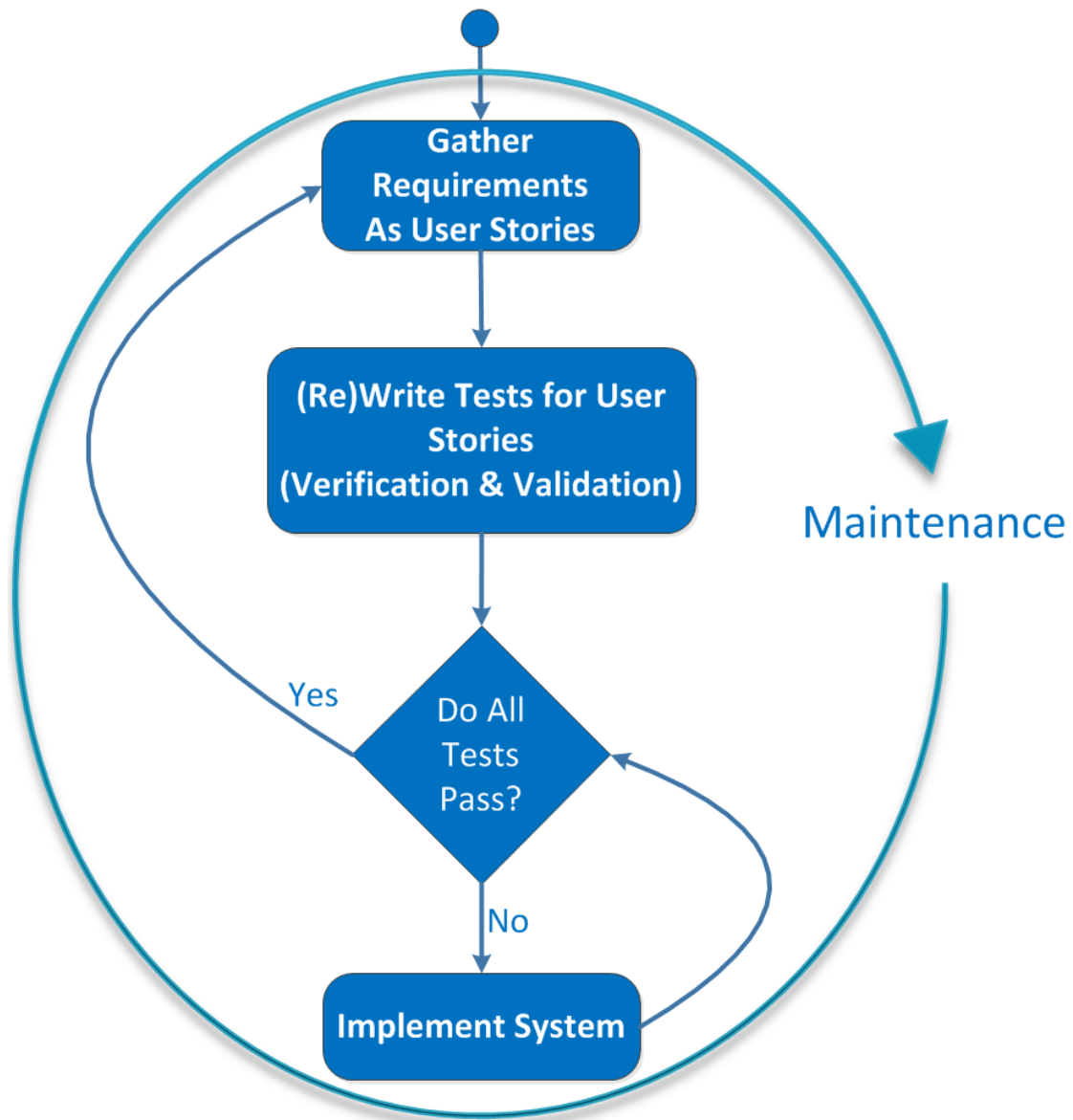


Figure 1.4. Steps in Test Driven Development [76, 23, 89]. Requirements are gathered via user stories first. Then tests are implemented to capture them. Afterwards, the production system is implemented until all tests pass. Once an iteration is complete, the system is expanded with new user stories in upcoming iterations until the whole system is developed.

are realized in the system, until the final system is developed. The advantage of these processes is that they are very responsive to changing requirements, and they provide partially or completely working systems in increments. In these processes, the incrementally built system itself serves as the documentation, which can be considered a disadvantage given that the process relies on the tacit knowledge embedded in the development team and system in response to rapidly changing requirements [29]. Some of the work in this dissertation (Chapters 4 and 6) is more applicable to agile processes due to the difference in the emphasis they put on the system implementation and test-cases. This is discussed in more detail in Section 2.2.4.

There are many software development processes that are a hybrid of these two approaches lying in different points of the spectrum between plan-driven and agile processes. An example hybrid process is the Spiral Model [31], where the development follows a plan-driven model, but in multiple iterations (see Figure 1.5). In each iteration, the scope of the project is expanded after reducing the risks for the next iteration, and the steps in the plan-driven model are repeated. Examples of other development processes are the Code-and-Fix model [108], Evolutionary Development [133, 108] and the V-Model [116, 56] which lie in the spectrum between plan-driven and agile processes.

Even though the software development process followed in a project might vary, the maintenance phase is typically a common and critical step for all processes. Maintenance includes any and all changes that need to be made on the developed system. The reasons for maintenance include: changing requirements and updating the system according to the updated requirements, adding new requirements, fixing bugs, and refactoring the system to improve its internal structure. The importance of the maintenance phase lies in its inevitability once the system is deployed and used by the stakeholders, since they will typically encounter bugs in the system or request for modifications on the system.

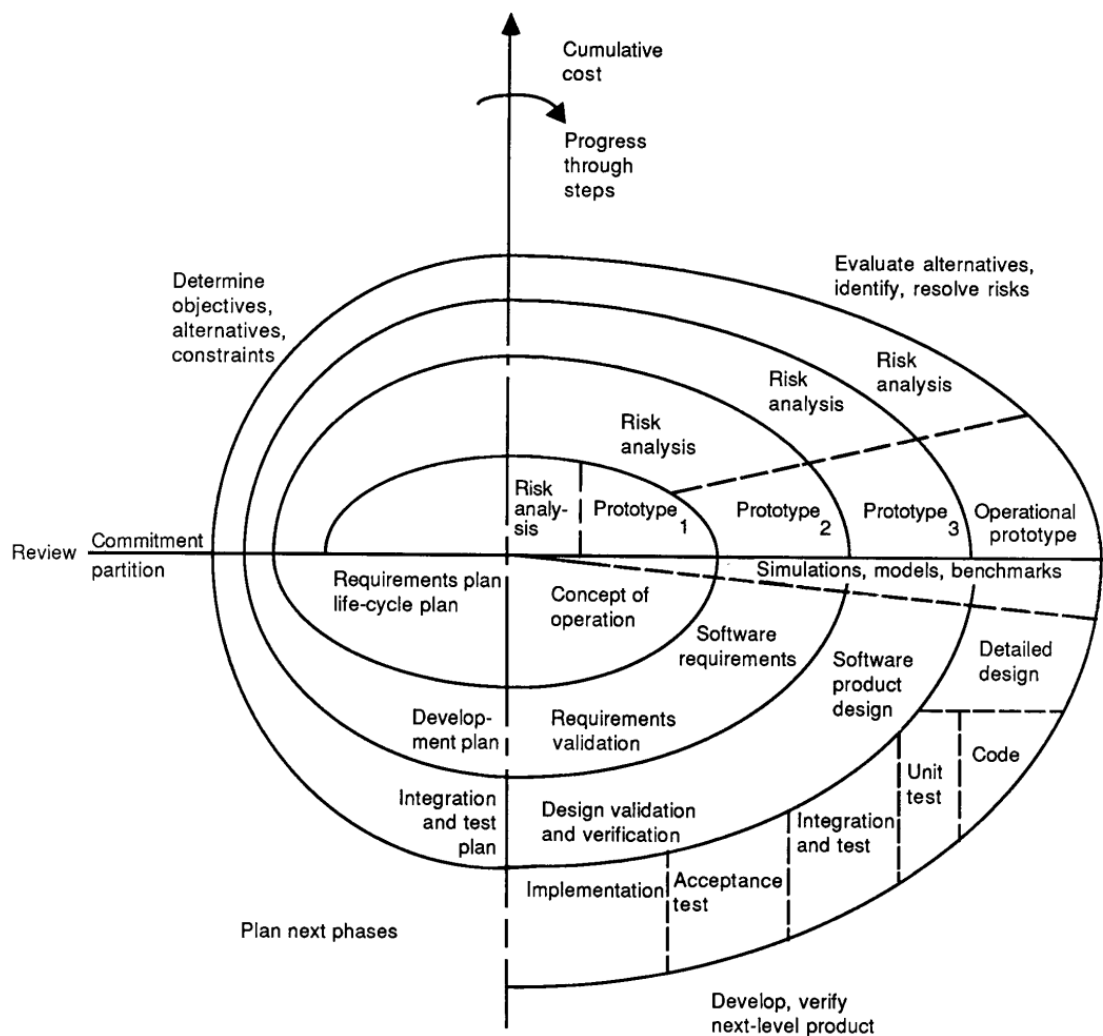


Figure 1.5. Phases of the Spiral Model, courtesy of Boehm [31]. The development moves from one phase to the other as in the Waterfall Model [127]. Iterations provide increments that expand the scope of the system while reducing risk.

1.2 Software Maintenance: the Most Costly Activity in the Software Development Lifecycle

In the lifecycle of a software system, maintenance is the process of modifying a software system or component after delivery to correct faults, improve performance or other attributes, or adapt to a changed environment [77].

According to the National Institute of Standards and Technology (NIST), software errors cost the United States (US) economy \$59.5 billion dollars annually [139]. Based on a research study published by the University of Cambridge, this number is \$312 billion dollars globally [90]. Software maintenance is the software development phase where these errors are fixed by developers.

Contrary to common belief, research suggests that software maintenance is the dominating cost in the lifecycle of a software system, with 70% —90% of the total cost [96], which Seacord et al. [132] named “legacy crisis”. Many factors affect the maintenance cost, including: misunderstood requirements, bugs introduced during development, insufficient testing, changing requirements, new requirements to be added to the system, and the time period the software product is in use.

The dominating activity in software maintenance is program understanding, with up to 60% of all cost of maintenance [43, 64]. Program understanding is the act of identifying different aspects of the system in relation to the existing, changing and new requirements. Before a maintenance task is performed, developers typically need to understand what the requirement is, and what part of the system needs to be changed. The changes to the system might include, and are not limited to: changing the design, the codebase, the test cases and the documentation. Therefore, software maintenance is strongly dependent on understanding requirements well.

1.3 Understanding Requirements is Important

Typically, requirements in a system change over time (update, deletion or addition of requirements). Therefore stakeholders constantly work on requirements, not only in the requirements gathering phase, but also during maintenance. The cost of changing a requirement increases dramatically over the lifecycle of a system [96, 30, 64]. While it is relatively cheap to correct errors due to misunderstood requirements in the earlier phases, it becomes prohibitively expensive towards the final stages of the lifecycle of the system. Therefore, understanding the requirements and obtaining as much information as possible about them becomes a major concern during maintenance.

Since maintenance usually comprises of a change in the requirements, it is critical for stakeholders to understand requirements before making any changes to the system, and using the knowledge about requirements on making critical decisions about the other steps of system development. This includes understanding important points such as:

- Change impact analysis: Determining which parts of the system would be impacted due to changes in requirements [22, 69, 121].
- Program comprehension: Understanding the relations between requirements (such as dependency) to help comprehension of the overall system [80, 117, 124, 153].
- Consistency checking: Determining if changes to the system have created unnoticed and unintended contradictions between requirements [33, 60, 91].
- Testing the system: Understanding requirements to identify the correct parts of the system to generate or reference appropriate test data, and to check if tests properly cover all requirements [22, 69, 144].
- Progress tracking: Monitoring the progress of the development of the system and the testing efforts [22, 69, 88, 121].

- Effort planning: Checking if development and testing efforts are invested correctly to align with the importance of requirements [36, 131, 136, 143].

To perform these activities quickly, either a developer that knows the system, or good documentation about the requirements and their relationships with the other artifacts (codebase, test cases, design documents) are needed. Even in the existence of good developers, it is typically beneficial to have requirements relationships documented and a linkage between requirements and other software artifacts, because developers typically work on different parts of the system at different times, and they might forget the knowledge acquired in a certain part of the system in the future [101, 102]. Therefore, if this type of information is not readily available in a system, it would be beneficial to mine this information from existing artifacts, such as tests.

1.4 Software Testing and Using Tests to Improve Maintenance

During the verification & validation phase, it is common to test different parts of the system to demonstrate the correctness with respect to requirements. Testing is an important part of the software development lifecycle, has many benefits for the final product, and is employed by many, if not all, software development teams. Based on empirical studies, in many systems, the amount of test code produced is comparable to the code produced for the system itself, ranging from 50 percent less to 50 percent more [107, 149]. Furthermore, based on a study conducted by NIST, more than a third of software errors in the US (corresponding to \$22 billion annually) can be eliminated with an improved testing infrastructure [139], which demonstrates the importance of testing and the investment on testing infrastructure in the software development lifecycle.

There are different types of testing, with the most common ones including:

- Unit testing: Testing a specific component, usually at the function level. These tests are typically implemented by the developers as they develop the system's components.
- Integration testing: Testing the interfaces and interactions of multiple components. These types of tests typically start with the interaction of a small number of components and expand to test the interaction of multiple components.
- System testing: Testing a completely integrated system to verify that it meets the requirements.
- Acceptance testing: Testing performed by the customers of the system to confirm it meets the requirements put forth by them.

Having many tests increases the effort spent on testing, and its cost in the development process. As Myers [111] points out, in 1979, in a typical programming project, approximately 50% of the total development time and more than 50% of the total cost of a system is expended for the testing phase. These numbers can be even higher for agile processes that put emphasis on tests, such as TDD [23, 89, 76].

Many software development teams use test results to determine if requirements are met. They run the tests and check the results to see if they pass or fail. However, tests contain more information that may be useful to stakeholders. Developers implement test cases to make sure a requirement, or the interaction of multiple requirements is met. Therefore, tests can be used as good sources of information about requirements and how they are linked to the artifacts (such as source code) in the system developed.

In this dissertation, we develop ways to understand what is captured in tests and exploit this information to better understand the relationships between requirements themselves and their relationship with tests. We form a convection cycle between

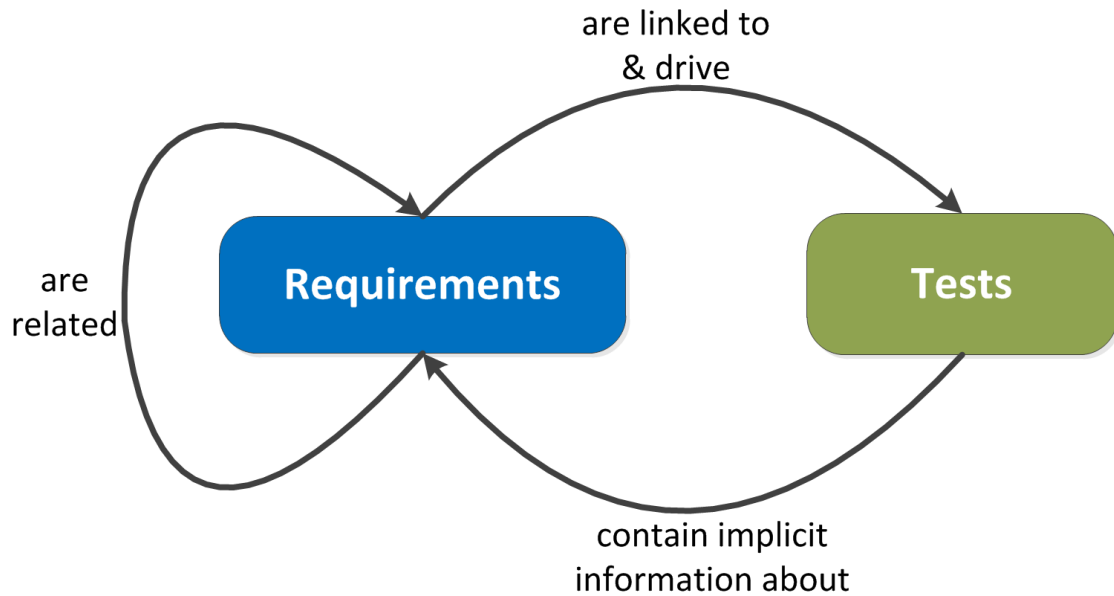


Figure 1.6. Convection cycle between requirements and tests.

requirements and tests (see Figure 1.6), so that stakeholders can benefit from tests during software maintenance activities from the requirements perspective, and maintenance related activities about requirements (such as the ones listed in the previous section), and do this in shorter time frames to reduce the overall maintenance cost of the system.

The work in this dissertation applies to all of these different types of testing on a high level. Specifically, the work in Chapters 4 and 6 can be considered to closely apply to unit, system and acceptance tests, since these types of tests verify the behavioral aspects of the system. The work in the other chapters apply to all types of testing described above.

In the next section, we discuss existing work on this topic, explain their limitations and describe where the work in this dissertation falls into.

1.5 Limitations of Previous Work

To form the convection cycle in Figure 1.6, existing techniques in the literature can be used. Figure 1.7 provides a high-level summary of these techniques, and where

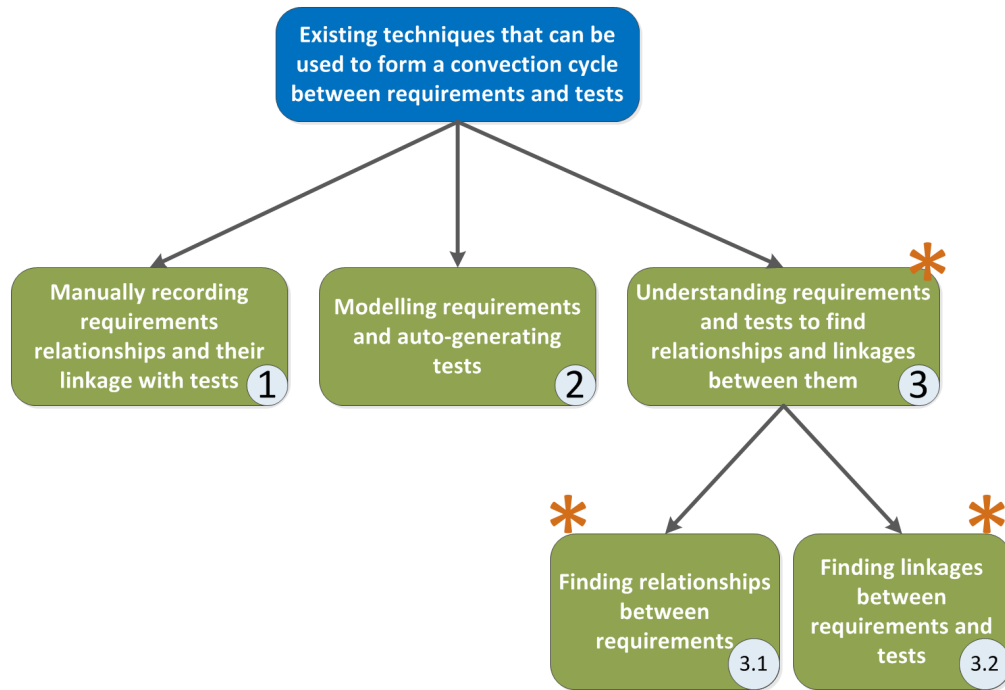


Figure 1.7. High level summary of the existing techniques to form a convection cycle between requirements and tests. * signs indicate where the work in this dissertation falls.

this dissertation stands compared to these techniques.

There are software development processes and tools that allow manually recording requirements, their relationships and the relationships between requirements and tests [6, 9, 12] (see 1 in Figure 1.7). In these systems, as shown in Figure 1.8, each requirement is typically given a unique identifier, and metadata can be stored for each requirement. Metadata can include the relationships between requirements and other artifacts (such as tests) [15, 150]. It is reported in the literature that these techniques are useful and successful, especially when they are customized for the organization using them [16, 17, 22, 54]. However, recording requirements related information this way is regarded as a hindrance [21] and reported to be error-prone and labor-intensive, and it typically requires disciplined developers [34, 69, 99]. Therefore, there are many systems built without using such techniques and tools to manage requirements. Furthermore, following such practices can be prohibitively expensive for legacy systems if they have not used

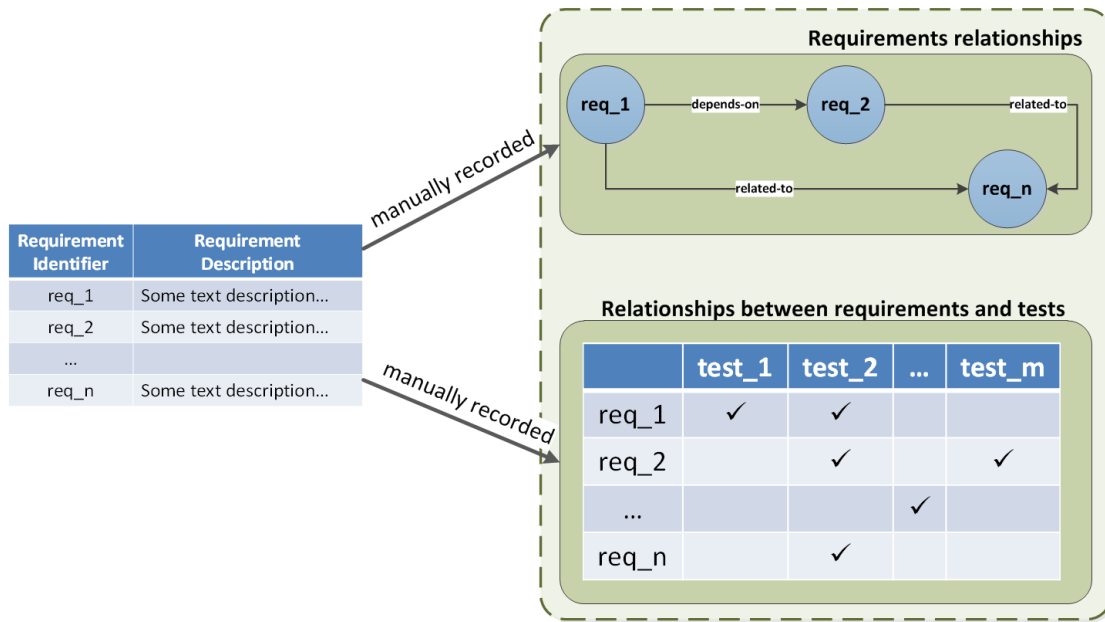


Figure 1.8. Recording relationships between requirements themselves and with tests manually.

these tools from the beginning.

Alternatively, some systems are built using a Model Driven Development approach [11], where semi-formal or formal models of requirements are created [48, 57, 70, 154, 112, 82, 38], and the artifacts (such as the source code and tests) are generated from these models fully or partially after formal model transformations [15, 150, 45, 46, 52, 142, 115] (see 2 in Figure 1.7). In such systems, as shown in Figure 1.9, the relationships between requirements can be recorded in/between the models themselves during the modeling phase. Such relationships can then be kept intact during model transformations, since the models are the basis of the developed system, and the meta information about requirements can be carried over as new models and artifacts are generated from the requirements models [15, 150]. Furthermore, if tests are also automatically generated at the end of a series of model transformations, linkages between requirements and tests can be maintained and used during maintenance [65]. Since the relationships between requirements and tests are automatically obtained in MDD, not

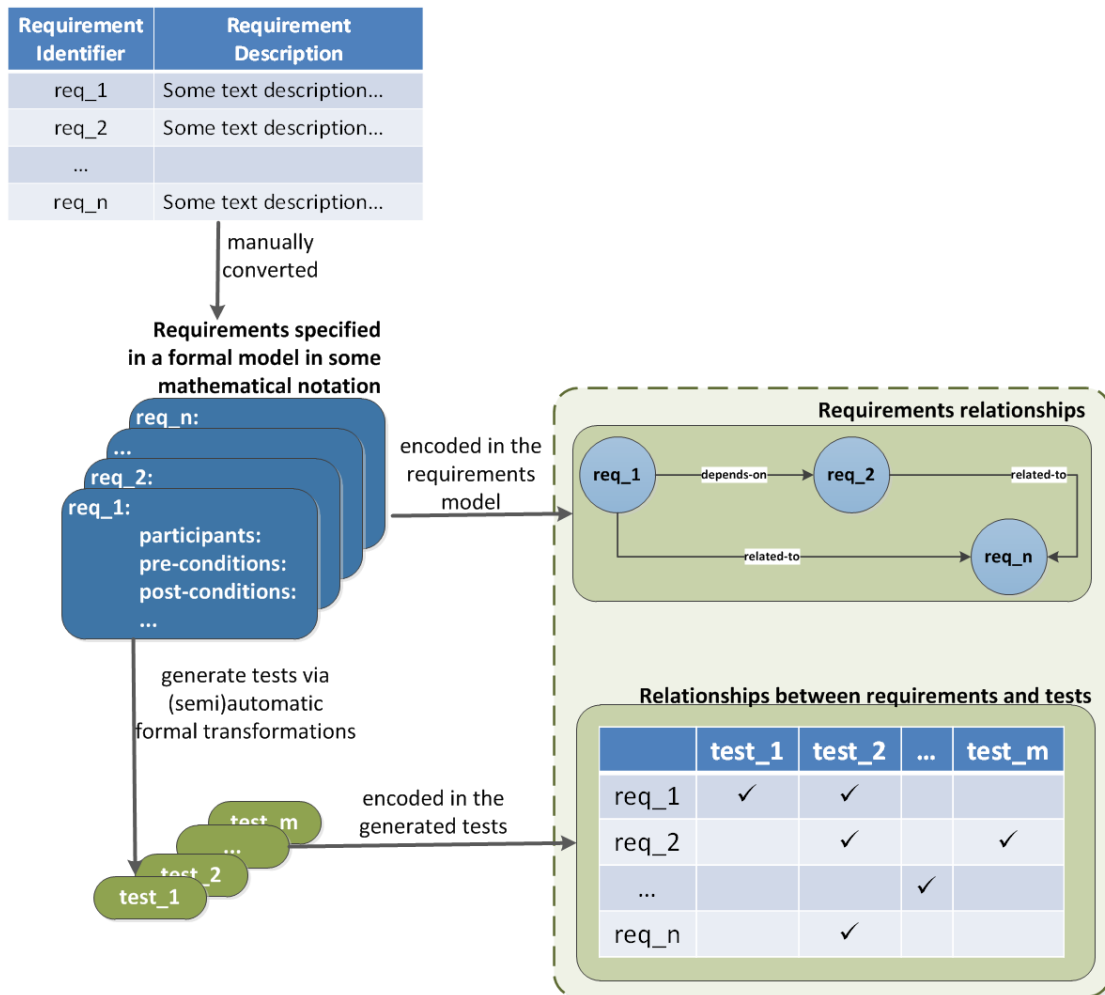


Figure 1.9. Getting relationships between requirements themselves and with tests automatically or semi-automatically using MDD [11].

much effort is required from stakeholders. However, many software projects do not follow an MDD approach. One reason for this is because not all project teams want to use a formal notation to describe requirements. Also, the abstractions required by MDD may not suit the needs of some systems. Furthermore, legacy systems developed without using models will not have these information neither. For such projects, other means are needed to retrieve and record the relationships between requirements themselves and with tests.

For systems that do not have requirements relationships information (with either

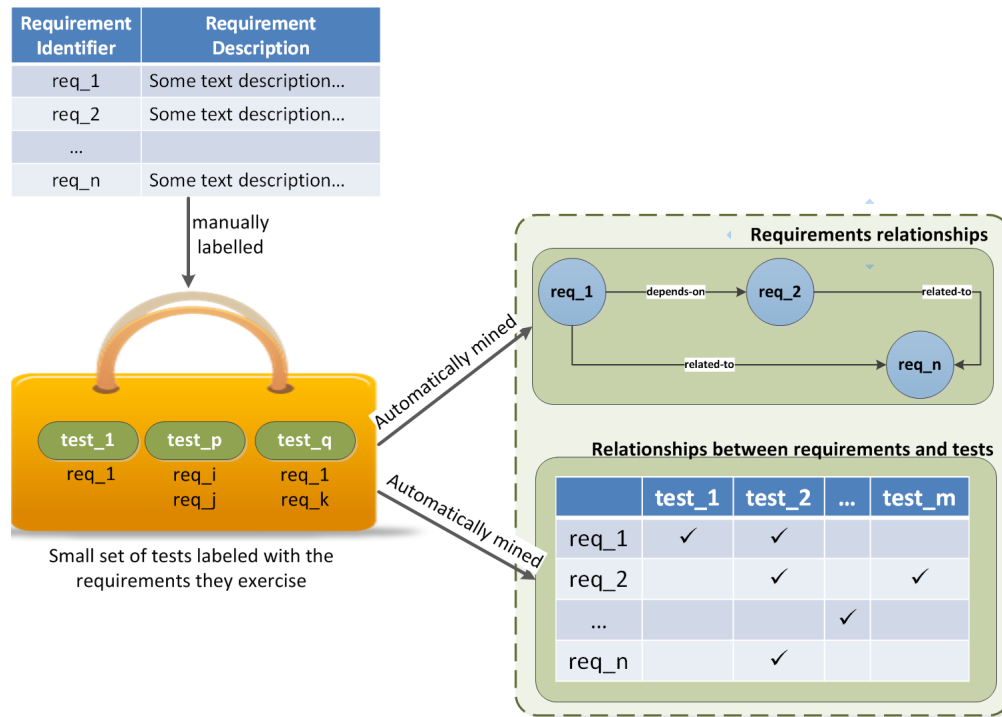


Figure 1.10. Mining relationships between requirements themselves and with tests by mining them from the existing tests.

of the techniques discussed above), for legacy systems and for systems built without following an MDD process, requirements relationships and their relationships with tests can be mined from tests if the system already has a test suite. One could understand requirements and tests, and link them together as shown in Figure 1.10. This would help exploit the information in legacy systems with existing tests, new systems built without modeling the requirements and without using test generation, by forming a convection cycle between requirements and tests. In this dissertation, we propose finding the relationships between requirements, and their relationship with tests for such systems. These techniques are classified as 3 in Figure 1.7.

Further breaking down the family of techniques in 3 in Figure 1.7, techniques exist to understand the relationships between requirements, and their relationship with tests.

First, there exist techniques to find relationships between requirements (see 3.1 in Figure 1.7). The first family of techniques use dynamic analysis, i.e. they run the system and examine its behavior while it is executed. These techniques target Object Oriented (OO) systems, where data in the system is encapsulated as objects [130, 95]. They exercise requirements on the existing system, and track the objects flowing through different parts of the system as they are being exercised. After further analysis of these objects, they determine relationships between requirements. These techniques only target Object Oriented systems since they follow object references flowing through the system, and they are highly sensitive to the system implementation and may suffer due to low coverage, a common problem with dynamic analysis methods where only the parts of the system that are exercised can be examined, while the other parts might also contain useful information. The second family of recent effective approaches find requirements relationships by executing them on the system, and analyzing the execution traces (e.g. method/class names) gathered while they execute [60, 117]. These techniques are not limited to object oriented systems, unlike the previous ones. However, they find only a small subset of requirements relationships, they are very sensitive to the way requirements are executed on the system, and they may also suffer from low coverage. These techniques are further discussed in detail in Chapter 6.

Then, there are recent effective techniques to link requirements with tests [18, 100, 104, 75, 98, 105] (see 3.2 in Figure 1.7). They rely on the analysis of textual information between the descriptions of requirements and the test code using Information Retrieval (IR) techniques. They exploit the similarity of the domain terms used in describing requirements and the terms used in test code, in comments and names. Even though there are recent techniques to improve their effectiveness [156, 109], these techniques achieve low accuracy overall due to the noise involved in natural language processing. These techniques are further discussed in detail in Chapter 4.

In this dissertation, we build upon existing techniques in the literature (specifically from the field of 'feature location') as the foundation of the work we propose. Building upon these techniques, we first improve the existing techniques (described above) to achieve better accuracy. Then we build upon them to find requirements relationships, and form a convection cycle between requirements and tests to aid software maintenance tasks. The work in this dissertation falls in 3 in Figure 1.7.

1.6 Main Contributions and Dissertation Outline

As discussed in the previous sections, maintenance is the most costly activity in the software lifecycle. Maintenance usually deals with changing or new requirements. Therefore it is vital for developers to understand requirements, their relationships with each other, and how they are linked to different system artifacts before they perform a maintenance task. In this dissertation, we use test cases as a valuable source of information on requirements to aid maintenance tasks.

Figure 1.11 shows an outline of this dissertation. The numbers in Figure 1.11 correspond to the chapter numbers that discuss the original work done on each task in detail.

The initial step is to find linkages between requirements and source code, i.e. where requirements are implemented in the source code, called requirements tracing/feature location in source code. In this work, discussed in Chapter 3, we improve an existing technique to make feature location a repeatable and end-to-end process in the software development lifecycle by making use of existing tests of the system.

Next, we use these source code linkages to identify linkages between requirements and test cases, i.e. which test cases exercise which requirements, called requirements tracing in tests. The technique we propose, discussed in Chapter 4, improves upon existing techniques by increasing the accuracy of obtaining the traces and increasing the

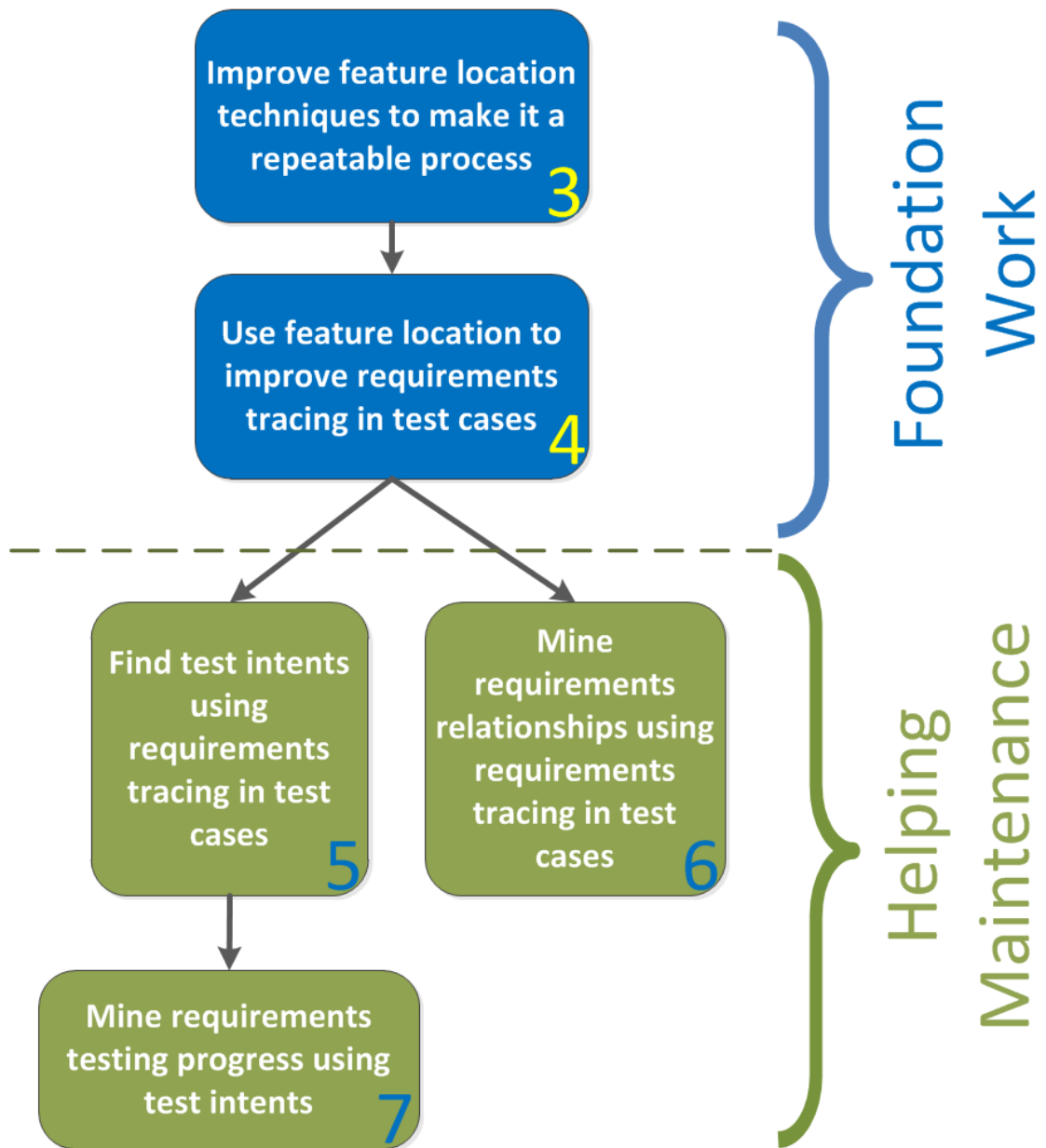


Figure 1.11. Outline of the dissertation. Numbers in each box are the chapter numbers where each work is explained in detail.

coverage of the found links.

The work in these two chapters is the foundation work that enables the rest of the chapters to form the convection cycle between requirements and tests. Once the linkages between requirements and test cases are available, we can gather the implicit information about requirements that exist in the test cases to aid maintenance.

In Chapter 5, we answer the question: What requirement(s) is a test case implemented for? This work builds upon existing techniques in the literature to enhance the semantics of requirements tracing links, so that stakeholders can quickly identify the actual requirement(s) tests are targeting to test.

In Chapter 6, we propose a new technique to mine relationships between requirements from tests automatically. These relationships allow stakeholders on determining requirements level change impact and consistency checking.

Finally, in Chapter 7, we propose a new requirements level view on the testing phase of the development of a system. We answer the question: How well does the test suite of a system cover its requirements? Answering this question helps different stakeholders, especially managers and developers on progress tracking of the testing efforts.

Chapter 2

Running Example and Background

In this chapter, we provide a running example on which to demonstrate the techniques used and proposed. We also introduce some terms that are commonly used throughout this dissertation and show examples to some of the terms on the running example.

In this dissertation, software system, software product, system and product are used interchangeably and they refer to a software system that is built to provide a service to its stakeholders along with all the artifacts that are typically associated with such systems, such as requirements specifications, design and architecture documents, source code, configuration files, test code, system documentation and user documentation.

2.1 Running Example

Throughout this dissertation, we use an example system to demonstrate concepts and show sample output on it for the techniques discussed. This example is the simplified version of a software system used in the CSE 70 Software Engineering class, offered in the Winter 2010 quarter at the University of California, San Diego (UCSD) by Professor Ingolf Krüger. Note that we also use the actual Chat System taught in UCSD as a case study to evaluate our techniques throughout the dissertation. The running example we use is only a simplified version of it.

In this system, students implement different portions of an end-to-end Chat System including the server and the client, so that they can send chat messages to each other over a network.

2.2 Background

In this section, we introduce some terms used throughout this dissertation, and provide samples on our running example.

2.2.1 Program Analysis Techniques

Once a software system is partially or completely developed, it can be analyzed automatically, called program analysis. Program analysis has many benefits including correctness, optimization, verification and performance measuring [113]. Program analysis techniques are broadly categorized as static analysis and dynamic analysis.

Definition 2.2.1 (Static Analysis). Static analysis is the process of evaluating a system or component based on its form, structure, content, or documentation [77].

Definition 2.2.2 (Dynamic Analysis). Dynamic analysis is the process of evaluating a system or component based on its behavior during execution [77].

Static analysis is used to analyze the system artifacts such as source code statically, i.e. without executing the system. Since the system is not executed, the analysis is on the structure and content of the system. An example to static analysis is to investigate the text in the comments of the source code to see if the terms used in comments and variable names correspond to the terms used in the initial documentation of the system.

On the other hand, for dynamic analysis, the system is executed to gather information while some test input is exercised, and the gathered information is later analyzed. An example to dynamic analysis is to obtain code coverage [110], where the system is

monitored while tests are executing to understand which lines of the source code are executed by the tests.

Some chapters of this dissertation (Chapters 3, 4, 6) use dynamic analysis, while the rest uses static analysis (Chapters 5, 7).

Another categorization of program analysis techniques is based on the focus of the analysis technique: data flow analysis or control flow analysis.

Definition 2.2.3 (Data Flow Analysis). Data Flow Analysis analyzes the sequence in which data transfer, use and transformation are performed during the execution of a computer program [77].

Definition 2.2.4 (Control Flow Analysis). Control Flow Analysis analyzes the sequence in which operations are performed during the execution of a computer program [77].

Data flow analysis focuses on the flow of data, such as the variables passed to functions or objects (in OO systems) passed around the program. On the other hand, control flow focuses on the sequence of calls in the program, such as which function is called by which function in the program. In this dissertation, the chapters that use dynamic analysis (Chapters 3, 4, 6) use control flow analysis.

2.2.2 Requirements

Software development typically starts with a discussion on what the system should provide to its stakeholders, i.e. its requirements.

Definition 2.2.5 (Software Requirement). The IEEE Standard Glossary of Software Engineering Terminology [77] defines a software requirement as: **(1)** a condition or capability needed by a user to solve a problem or achieve an objective, **(2)** a condition or capability that must be met or possessed by a system or system component to satisfy a

contract, standard, specification or other formally imposed documents, **(3)** a documented representation of a condition or capability as in (1) or (2).

Intuitively, requirements are what a system needs to provide to its stakeholders, i.e. what the customer of a software system expects to be delivered at the end of development. As described by Brooks, *“the hardest single part of building a software system is deciding precisely what to build. No other part of the conceptual work is as difficult as establishing the detailed technical requirements...No other part of the work so cripples the resulting system if done wrong. No other part is as difficult to rectify later.”* [35] Therefore, requirements play an important role in all stages of the lifecycle of the system development. For managers, requirements provide a general overview on what needs to be developed and provided to customers, as well as a way to measure development progress. For designers, requirements define the specification and constraints on the behavior and qualities of the system. For developers, they define the acceptable system behavior as well as other non-behavioral qualities of the system (such as reliability, security and performance). For testers, they define a basis to verify and validate the system built. Requirements can even be used for marketing to highlight the system’s important properties [63].

Requirements are typically documented in requirements specification documents.

Definition 2.2.6 (Requirements Specification Document). Requirements specification document is a document that specifies the requirements for a system or component [126].

In the requirements specification document, requirements are typically listed with a unique name or identifier, and a description. There are different ways to create and maintain a requirements specification document. While tools such as DOORS [6] and RequisitePro [6] can be used, it is also common to use web pages or text documents for this purpose. Table 2.1 is the sample requirements specification document for our running

Table 2.1. Requirements specification document for our running example Chat System. Each requirement is assigned a unique identifier, a name and a description.

ID	Requirement Name	Requirement Description
1	connect	Users should connect to the server before they start doing anything else. Connect operation connects to the backend server and gets a listing of all users currently connected.
2	sign-on	After they connect, users should sign on to the backend server with their credentials. The backend server will return a success or failure message to the client upon attempt to sign on.
3	send-message	Once they sign on to the backend server, users can send messages to each other. Messages should have a timestamp so that they can be ordered by the backend server.
4	sign-off	Once users are done using the system, they can sign out of the backend server. This will end their session, and free up resources in the backend server. Only users that have signed on successfully can sign off.

example. Each requirement has a unique identifier, a name and a textual description.

Requirements are further classified into two general categories depending on whether they are behavioral or qualitative: functional and non-functional requirements.

Definition 2.2.7 (Functional Requirement). A functional requirement is a requirement that specifies a function that a system or system component must be able to perform [77].

Definition 2.2.8 (Non-Functional Requirement). A non-functional requirement is a qualitative property of the system [50, 86, 79, 67] that restricts the types of solutions one might consider [126].

The functional requirements capture the nature of the interaction between the component and its environment [126]. They are the behavioral services the system provides to stakeholders, i.e. *what* the system should *do*. Non-functional requirements describe the quality attributes of the system, such as security, fault-tolerance, scalability,

Table 2.2. Features and scenarios for each feature for the Chat System

Feature Name	Scenario
connect	On the GUI, click “Connect to server”.
sign-on	Enter your credentials in the provided input boxes and click “Sign on”.
send-message	On the GUI, double click on a friend’s name, type some text in the message box that opens and click “Send”.
sign-off	On the GUI, click “Sign off”.

maintainability, and testability. Therefore, they describe *how* the system should *be*.

The requirements in Table 2.1 are all functional requirements. And if we were to have a requirement that reads: “Backend server should be scalable to support a thousand users simultaneously”, this would be classified as a non-functional requirement. Some parts of this dissertation focus only on functional requirements, while other parts apply to both types of requirements. Either way is explicitly specified in the relevant context.

2.2.3 Features

Once the requirements of the system are gathered, the system is developed according to those requirements. The realization of requirements in the system are features.

Definition 2.2.9 (Feature). Features are defined as behaviors of the system observable by users during their interaction with the system [62].

Functional requirements of a system are realized by features. This means that features are defined in systems where development already took place (at least partially), and behavior that corresponds to requirements exist in partial or complete form. Features can be triggered by the users of the system [61]. For instance, once the Chat System is implemented, it will have the features listed in Table 2.2 that correspond to the requirements listed in Table 2.1. Figure 2.1 shows an overview of the relationship

between requirements and features as they are described here. In our Chat System example, features that exist in the system map one-to-one to the requirements of the system, while there are also non-functional requirements of the system, which are not captured by the features as we define them here.

Since features exist in the system when it is partially or completely implemented, there must be a way to activate/trigger them. This is where scenarios are used.

Definition 2.2.10 (Scenario). A scenario is a sequence of steps, which a stakeholder has to perform on a software system in order to exercise a feature of interest [55]. A scenario defines the context in which a feature is studied, for example the sequence of the developer's actions with the program [119].

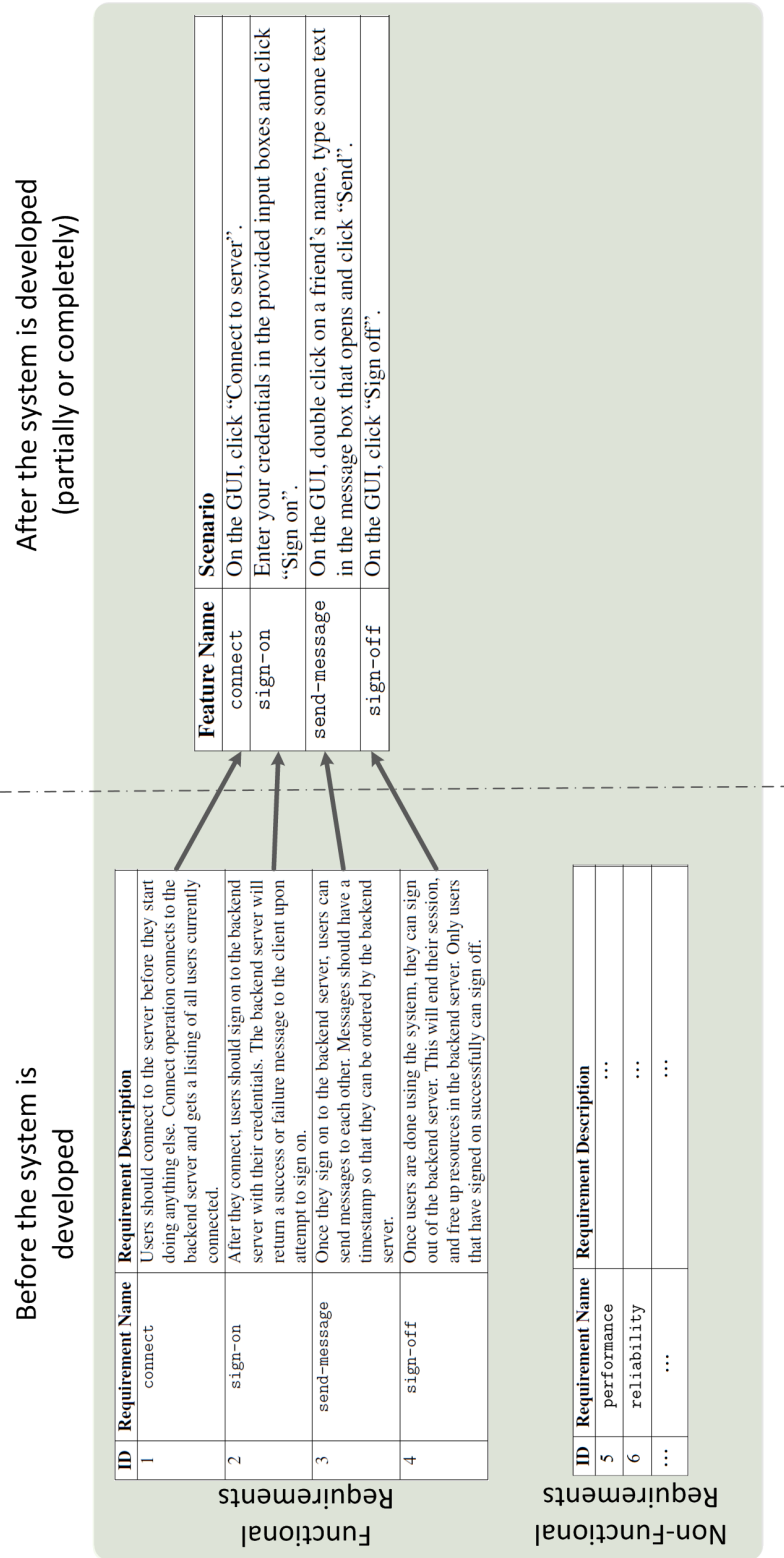


Figure 2.1. The relationship between requirements and features. In our Chat System example, features that exist in the system map one-to-one to the requirements of the system, while there are also non-functional requirements of the system, which are not captured by the features as we define them here.

```
student.client.CommandParser.sendText [CommandParser.java:102]
```

Figure 2.2. Sample execution unit for Java [68]. The initial number is the call depth, while the rest is a concatenation of the class name and the method name, along with the location of the method in the source code.

Intuitively, a scenario for a feature is a sequence of actions that trigger the feature on the system. In our Chat System, if there is a graphical user interface (GUI) provided, scenarios can be described for all features as in Table 2.2. There are other ways to create scenarios. Scenarios can be programs implemented using application programming interfaces (APIs). They can also be test cases from the system’s test suite (unit test, system test), or providing any other executable specification that might represent a requirement, such as a formal test specification maintained by tools like FIT [8].

As scenarios are executed on the system, stakeholders can gather information on the locations in source code that were exercised by using instrumentation. Instrumentation is a technique where the running software system is monitored while it is running to gather information on what locations of the system are being executed. The next two definitions describe these locations.

Definition 2.2.11 (Execution Unit). An execution unit is a source code entity of desired granularity. Examples are statements, blocks, methods/functions, classes/files and packages.

Execution units typically contain the exact location of the executed portion of the source code, such as file name and line number, as shown in Figure 2.2. Execution units are collected while scenarios or tests are executing on the system, i.e. during dynamic analysis. In this dissertation, we used the combination of classes and methods as execution units. However, the granularity can be changed to have higher or lower resolution, to accommodate different programming languages and project properties. For a sample execution unit, see Figure 2.2.

When a scenario is executed on a running system, there will be many execution units triggered in the system. The collection of these execution units is called an execution trace.

Definition 2.2.12 (Execution Trace). An execution trace for a scenario is the (possibly ordered) list of execution units observed while the scenario is executed on the system.

Execution traces are typically collected for dynamic analysis, and they can be collected in different ways, such as using a profiler, through instrumentation, or using AspectJ [5] for Java [68]. For a sample execution trace, see Figure 2.3, which shows the execution trace for the send-message feature from the Chat System. In this execution trace, the initial numbers show the call depth of the execution unit, i.e. it encodes the caller-callee relationship between the execution units. As an example, a depth of 0 indicates that the program started executing the first execution unit, and that in turn calls other execution units (methods from other classes in our case). Therefore, the called execution units have depth greater than 0.

2.2.4 Requirements and Features: Connecting the Dots

In Sections 2.2.2 and 2.2.3, we made the distinction between requirements and features, where requirements describe what the system should do or how it should be, while features exist in the system and they are the realization of the functional requirements. As shown in Figure 2.1, to precisely identify which requirement is related to which feature, one needs to have a mapping between the two. This mapping is typically implicit knowledge for the stakeholders developing the system, since they go through what is described in requirements and build the system to provide features for those requirements. However, other than this implicit knowledge, we may not have an explicit representation between the two.

```

[3]      infrastructure.router.NodeProxyJMS.dispatch[NodeProxyJMS.java:198]
[4]      infrastructure.router.NodeProxyJMS.getDestination[NodeProxyJMS.java:156]
[5]      global.addresses.NodeAddress.getNodeID[NodeAddress.java:28]
[4]      infrastructure.router.NodeProxyJMS.getDestination[NodeProxyJMS.java:156]
[5]      global.addresses.NodeAddress.getNodeID[NodeAddress.java:28]
[4]      infrastructure.router.NodeProxyJMS.getDestination[NodeProxyJMS.java:156]
[5]      global.addresses.NodeAddress.getNodeID[NodeAddress.java:28]
[3]      infrastructure.messages.InfrastructureMessageJMS.getPayload[InfrastructureMessageJMS.java:27]
[3]      student.client.Client.dispatch[Client.java:69]
[4]      global.messages.CommandMessage.getCommandString[CommandMessage.java:21]
[5]      global.messages.Message.getPayload[Message.java:30]
[5]      global.messages.payloads.StringMessagePayload.getContent[StringMessagePayload.java:17]
[4]      student.client.CommandParser.parseCommand[CommandParser.java:46]
[5]      student.client.CommandParser.sendText[CommandParser.java:102]
[6]      student.client.Client.sendText[Client.java:214]
[7]      student.client.Client.isSignedOn[Client.java:251]
[7]      student.client.Client.send[Client.java:111]
[8]      infrastructure.router.NodeProxyJMS.send[NodeProxyJMS.java:174]
[9]      infrastructure.router.NodeProxyJMS.getDestination[NodeProxyJMS.java:156]
[10]     global.addresses.NodeAddress.getNodeID[NodeAddress.java:28]
[9]     infrastructure.router.NodeProxyJMS.getDestination[NodeProxyJMS.java:156]
[10]     global.addresses.NodeAddress.getNodeID[NodeAddress.java:28]
[9]     infrastructure.router.NodeProxyJMS.getDestination[NodeProxyJMS.java:156]
[10]     global.addresses.NodeAddress.getNodeID[NodeAddress.java:28]
[3]      infrastructure.messages.InfrastructureMessageJMS.getPayload[InfrastructureMessageJMS.java:27]
[3]      student.client.Client.dispatch[Client.java:69]
[4]      student.client.Client.isMessageWellFormed[Client.java:282]
[5]      global.messages.Message.getSource[Message.java:22]
[5]      global.messages.Message.getDestination[Message.java:26]
[5]      global.messages.Message.getPayload[Message.java:30]
[4]      student.client.Client.sendLocalDisplayMessage[Client.java:101]
[5]      global.utility.Observer.dispatch[Observer.java:25]
[6]      student.client.TUI.TextView.dispatch[TextView.java:14]
[7]      global.messages.LocalDisplayMessage.toString[LocalDisplayMessage.java:15]
[8]      global.messages.Message.getPayload[Message.java:30]
[8]      global.messages.Message.getPayload[Message.java:30]

```

Figure 2.3. Sample execution trace of the send-message feature of the Chat System. The caller - callee relationships are reflected in the execution trace with tabs.

Furthermore, as Wiegers describes, “a feature is a set of logically related functional requirements that provides a capability to the user and enables the satisfaction of a business objective” [147]. Based on this description, a requirement and a feature may or may not have a one-to-one relationship where a single feature maps to a single requirement. In our Chat System example, they map to each other one-to-one (as shown in Figure 2.1). However, this will not be the case for all systems.

The mapping between requirements and features typically surfaces differently in the different software development processes, mostly due to the emphasis the processes put on the different parts of the system, as discussed in Section 1.1. For plan-driven processes, since requirements are largely determined in the beginning stages of the development lifecycle, the relationship between requirements and features may not be one-to-one. For agile processes, however, the requirements are typically encoded as user stories, for which features are implemented as the system is developed. As an example, in TDD, first tests are implemented to represent user stories, which encode requirements (as shown in Figure 1.3). Then, features corresponding to these requirements are implemented in the system. Therefore, in agile systems, the relationship between requirements and features can be considered to be closer to one-to-one. Several chapters in this dissertation (Chapters 4 and 6) describe work on the premise that requirements and features have an almost one-to-one relationship. Therefore, the work in those chapters are expected to be more applicable to agile processes.

Although the scope of the work in this dissertation is partially limited to agile processes, it can be expanded to apply to plan-driven processes too. For this, the techniques discussed across this dissertation can be complemented with an additional step that lets stakeholders provide the mapping between requirements and features (albeit it is a one-to-many relationship). Overall, the techniques discussed can be extended to use this mapping to take in the proper inputs and provide the proper outputs for requirements

and features separately. This is left as a benefit considered within the scope of future outlook provided by this dissertation.

2.2.5 Requirements Tracing

Since software development is driven by requirements, the developed system artifacts will have linkages back to the requirements. Requirements Tracing aims to find those linkages.

Definition 2.2.13 (Requirements Tracing). Requirements Tracing or Requirements Traceability (RT) is defined as the ability to describe and follow the life of a requirement, in both a forward and backward direction [18], by defining and maintaining relationships to related development artifacts [100], such as software architecture documents, design models, source code, test cases and configuration files.

For stakeholders of a system, RT provides many benefits such as prioritizing requirements, estimating change impact on code, proving system adequacy, validating, testing and understanding the system, and finding reusable elements [69, 137]. Recent studies also demonstrate quantitative evidence on the benefits of requirements tracing [101, 102], where it was determined that developers heavily relied on requirements tracing when it is available, requirements tracing has a great impact on the quality and performance of the developers' task of investigating source code, and developers were able to adopt requirements tracing quickly without much training.

Requirements impact the whole development phase, and during the development phase, stakeholders produce traces in the artifacts produced in the system that can be linked back to requirements. These traces are called requirements traces.

Definition 2.2.14 (Requirements Trace). The IEEE Standard Glossary defines a trace as a relationship between two or more products of the development process [77]. Intuitively,

Table 2.3. Sample Requirements Traceability Matrix for tests of the Chat System

	connect	sign-on	send-message	sign-off
t_1 : testConnect	✓			
t_2 : testConnectAndSignOn	✓	✓		
t_3 : testSendMessage	✓	✓	✓	
t_4 : testSignOff	✓	✓		✓

requirements traces are linkages between the requirements and other artifacts of a software system, such as source code, test cases and documentation.

Based on this definition, there can be requirements traces between requirements and source code, requirements and test cases, and similarly between requirements and any other types of artifacts of the system.

There are many ways to represent requirements traces, such as matrices [49, 145], databases [78], hypertext links [16], graphs [118] and formal methods [42]. In this dissertation, we frequently make use of a Requirements Traceability Matrix (RTM).

Definition 2.2.15 (Requirements Traceability Matrix). Requirements Traceability Matrix is a matrix that shows the traces between requirements and a type of system artifact.

In the RTM, columns typically contain requirements and rows contain another artifact, such as tests or source code components. The contents of the matrix shows the traces between the two. As an example, Table 2.3 shows the RTM between the tests and the requirements for our Chat System. The rows of the table are the test cases in the test suite, the columns are the requirements of the Chat System, and the ticks are the requirements traces between the two. Here, the requirements traces show which test cases test which requirements in the system.

2.2.6 Feature Location

As discussed in Section 1.2, to perform maintenance, developers typically identify the parts of source code that are related to a specific feature. This identification process

is performed using feature location [148].

Definition 2.2.16 (Feature Location). Feature location is the activity of identifying an initial location in the source code that implements functionality in a software system [26, 120].

With feature location, developers typically identify specific constructs in source code, such as classes or methods, that are highly relevant to a feature and use them as starting points for further investigation and program understanding to perform a maintenance task.

As discussed in Section 2.2.3, features are the realization of functional requirements of a system. Therefore, the feature location field has a close relationship with requirements tracing [60]. As discussed in Section 2.2.4, if requirements and features have a relationship that is close to one-to-one, such as in agile processes, the acts of requirements tracing and feature location have a very high overlap, especially on the source code level. In this dissertation, we exploit this overlap in several chapters.

2.3 Case Studies

Throughout this dissertation, we use case studies to evaluate the effectiveness of the techniques we propose by performing experiments. Most of these case studies are real systems used in production. Therefore, this dissertation is closely related to the field of empirical software engineering.

The properties of the case studies we use are listed in Table 2.4. We picked these case studies such that they are from different domains and can be representative of different types of software systems in production.

Table 2.4. Properties of the case studies used in this dissertation.

Case Study	# Lines of Code	# Lines of Test Code	Ratio of Source Code to Test Code	# Features / # Requirements	# Test Cases
Chat System	6861	3257	0.47	16	20
Apache Pool [3]	12626	8690	0.69	16	77
Apache Log4j [2]	52886	15952	0.30	10	69
Apache Commons CLI [1]	4739	3853	0.81	11	181

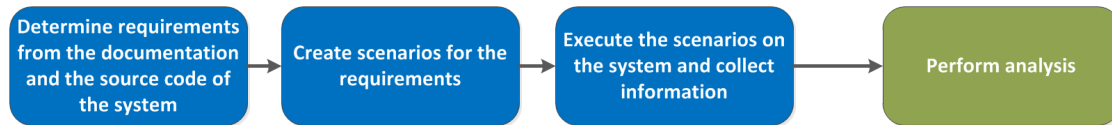


Figure 2.4. The overall flow of our activities to analyze our case studies.

The first case study, the Chat System, is used in teaching a class. Therefore, it has some ideal properties such as very good and thorough documentation, well defined requirements, a comprehensive test suite, test cases that map well to requirements and good documentation on which test cases are implemented to test which requirements. During the evaluation of our techniques, we expect to obtain better results on this case study compared to the other case studies. We picked the other case studies from open source projects. These are all active projects maintained by the open source community, and they are commonly used in production in many software systems. Some of these projects had good documentation, while others didn't. Due to these properties, we expected that they could demonstrate how well our techniques worked for systems with such different properties. Furthermore, we expected to find out how well the competing techniques worked on such different case studies, and how they were affected from their different characteristics. For instance, in Chapter 4, competing techniques use the documentation of the projects. We assess the effects of relying on documentation vs. using our approach (a dynamic analysis technique).

Finally, all of these systems are all implemented in Java [68], and they have test suites that run with JUnit [10]. Having a test-suite is a precondition for some of our techniques, so these case studies are very suitable for the work in this dissertation.

Figure 2.4 demonstrates the overall flow of our activities to analyze these systems. The upcoming sections discuss these steps in more detail.

2.3.1 Finding Requirements

Finding requirements/features in the case studies is the crucial first step in all of our case studies. The requirements in the Chat System were already documented with requirements specification documents in text form. We gathered the requirements in Apache Pool [3], Apache Log4j [2] and Apache Commons CLI [1] through their web pages, javadocs, and comments manually. At the end of this step, a requirements specification document similar to the one shown in Table 2.1 is obtained for each case study.

Note that it may not be possible to find all requirements in a software system through its documentation or source code. Therefore, we can only claim that we have performed our analysis on parts of the software in the case studies.

2.3.2 Creating Scenarios

Whenever we needed to perform dynamic analysis on our case studies (Chapters 4 and 6), scenarios are needed to trigger requirements on the system. For the Chat System we created scenarios manually. We brought up the chat server, and performed actions such as "connect" and "sign-on" using the existing graphical user interface of the system.

For Apache Pool [3], Apache Log4j [2] and Apache Commons CLI [1], we created scenarios as executable tests themselves using special markers, called annotations, in Java [68] (e.g. `@Scenario(requirement="")`). The tests obtained for these case studies are similar to the sample shown in Figure 3.6.

2.3.3 Collecting Execution Traces

Whenever we needed to perform dynamic analysis on our case studies (Chapters 4 and 6), we needed to gather execution traces of scenarios and tests while they are executed on the system. We collected the execution traces using AspectJ [5]. Note that

AspectJ is not a profiler, but it can be used for this purpose by weaving method entries and printing their names.

At the end of this step, outputs similar to the one shown in Figure 2.3 are obtained for each scenario.

2.4 Commonly Used Evaluation Metrics

In this section, we introduce metrics that are commonly used throughout this dissertation to evaluate the performance of several techniques.

2.4.1 Precision, Recall, F-Measure

Commonly used metrics to measure the quality of Information Retrieval techniques are *precision*, *recall*, and their combination *f-measure*. As shown in Figure 2.5, in such problems, the main target is to successfully find the true members of a set, i.e. the **relevant** set. A suggested technique will typically offer a set of items to belong to that relevant set, i.e. the **retrieved** set. The metrics precision, recall and f-measure indicate the success of the used technique from different perspectives. A simple example that demonstrates what these metrics correspond to is search engines, such as Google. When a user types in a query, the search engine retrieves a list of results that it considers related to the query. In this context, the relevant set is the list of links to all web pages that are relevant to the user's query. The retrieved set is the list of links that the search engine returns to the user.

There are three different possibilities of a member in relation to the relevant and retrieved sets. First, there may be false positives: instances that are retrieved by the retrieval technique, even though they are not in the relevant set. These belong to the set $retrieved \setminus relevant$.

Second, there are true positives: relevant instances that are retrieved correctly.

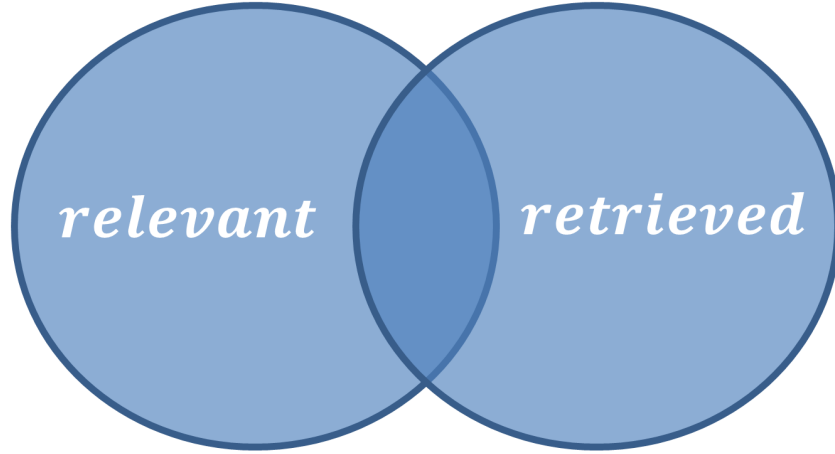


Figure 2.5. Sets describing precision, recall and f-measure.

These belong to the set $retrieved \cap relevant$.

Finally, there are false negatives: relevant instances that were not identified successfully by the retrieval technique. These belong to the set $relevant \setminus retrieved$.

Precision measures the accuracy of the retrieved links:

$$precision = \frac{|relevant \cap retrieved|}{|retrieved|} \quad (2.1)$$

On our search engine example, precision corresponds to the quality of the retrieved links, i.e. how many of them were actually relevant to the user's query.

Recall measures the completeness of the relevant links compared to the retrieved ones:

$$recall = \frac{|relevant \cap retrieved|}{|relevant|} \quad (2.2)$$

On our search engine example, recall corresponds to how many of the relevant links the search engine was able to find correctly.

For retrieval problems, either one or both of these metrics can be important. When they are both important, another metric is used to combine them into a single metric: f-measure. Precision and recall can be combined with different weights, but when they

are combined with equal weights, f-measure is defined as:

$$f\text{-measure} = 2 \times \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}} \quad (2.3)$$

In the rest of this dissertation, we make use of these metrics in several chapters to assess the success of techniques on retrieving artifacts such as requirements trace links.

2.5 Classification of the Work In This Dissertation

In the first two chapters, up to this point, we discussed different dimensions of software engineering and analysis, including:

- Plan-driven and agile software development processes,
- Static analysis and dynamic analysis,
- Data flow and control flow analysis,
- Functional and non-functional requirements.

To make the comprehension of the work in this dissertation easier, Table 2.5 provides an overview of the classification of our work in the dimensions above.

Table 2.5. Classification of the work in this dissertation in different dimensions.

	Type of Software Development Process That the Work Applies To		Type of Program Analysis Used				Type of Requirements That the Work Applies To	
	Plan Driven Processes	Agile Processes	Static Analysis	Dynamic Analysis	Data Flow Analysis	Control Flow Analysis	Functional Requirements	Non-Functional Requirements
Chapter 3		✓		✓		✓	✓	
Chapter 4		✓		✓		✓	✓	
Chapter 5	✓	✓	✓				✓	✓
Chapter 6		✓		✓		✓	✓	
Chapter 7	✓	✓	✓				✓	✓

Chapter 3

Feature Location Using Data Mining on Existing Test Cases

In this chapter, we present a novel technique that improves upon existing techniques to locate features in source code.

3.1 Introduction

Maintenance is one of the most costly and time consuming tasks in the lifecycle of a software system [96]. To perform maintenance, developers typically identify the parts of source code that are related to a specific requirement, i.e. feature location. With feature location, developers identify specific constructs in source code, such as classes or methods, and use them as starting points for further investigation and program understanding to serve a maintenance task. As an example, for our Chat System, source code location suggestions are presented for each feature in Figure 3.1. This is a listing of the highly relevant source code locations for each feature, ranked from the most relevant to the least. If a table similar to the one in Figure 3.1 is provided to a developer, she can perform maintenance tasks faster [101, 102]. She can start investigating the source code from the most relevant locations for a feature, and continue investigating the other parts of the source code using the structural relationships (e.g. code locations that call the most

Feature	Highly Relevant Source Code Locations (from more to less relevant)
connect	1. student.client.Client.isSignedOn (Client.java: 251) 2. student.client.Client.connectServer (Client.java: 126) 3. student.client.Client.isConnected (Client.java: 78) ...
sign-on	1. student.client.Client.getUser (Client.java: 244) 2. student.rooms.ChatRoom.addUser (ChatRoom.java: 23) ...
send-message	1. student.client.CommandParser.sendText (CommandParser.java: 102) 2. student.client.CommandParser.getUserAddress (CommandParser.java: 131) ...
sign-off	1. student.client.Client.signOff (Client.java: 255) 2. student.client.Client.setSignedOff (Client.java: 141) ...

Figure 3.1. Source code location suggestions for each feature in the Chat System. Feature location suggestions are ranked from more to less relevant. A developer can start investigating the implementation of a feature starting from these feature location suggestions.

relevant locations presented in Figure 3.1).

Feature location techniques can be classified based on different dimensions. Table 3.1 provides a categorization of the existing work in the literature based on two dimensions: type of analysis and interaction with user.

The first dimension is what type of analysis a technique performs: static analysis, dynamic analysis or a combination of the two. Static analysis based techniques analyze

Table 3.1. Classification of related work on feature location based on two dimensions.

	Automated	Interactive
Static Analysis	Robillard[123]	Biggerstaff [26] Robillard and Murphy [125]
Dynamic analysis	Wilde and Scully [148] Zhao et al. [155] Wong et al. [152]	Antoniol and Gueheneuc [19]
Hybrid	Poshyvanyk et al. [119]	Eisenbarth et al. [61] Liu et al. [97] Edwards et al. [59]

the static artifacts of the system (e.g. source code and configuration files) [26, 125, 123]. They typically exploit the structural relationships between the entities in source code (such as classes and methods) to identify more important entities. Then, when a developer needs to find locations for a specific feature, they suggest locations based on how important the entities in source code are, and how related they are with the requested feature based on textual cues (e.g. similarity in terms and naming conventions).

These techniques are known to be less effective [152] compared to dynamic analysis techniques, because they typically deal with a very large portion of the source code with a large amount of utility code, i.e. more noise. We refer the reader to [106] for a more detailed discussion of static analysis based approaches.

Dynamic analysis based techniques [148, 62, 19, 59, 152] exercise the system and collect information to later analyze and locate features. These approaches typically run the system, trigger features on it via scenarios, and collect information on which execution units were exercised while a feature was triggered.

There are also techniques that combine both static and dynamic analysis [61, 119, 97] to use information from multiple sources and provide better feature location suggestions to stakeholders.

Another dimension of feature location techniques is whether they are interactive or automated. Interactive techniques [125, 61, 39] require a feedback loop, where the user incrementally obtains more information about the location of a feature by interacting with the tool. Automated techniques [148, 155, 152], however, take in an initial input and perform automated or semi-automated analysis.

Our work is based on dynamic analysis and it is an automated technique. In the next section, we discuss dynamic analysis and hybrid (both dynamic and static analysis) techniques in more detail.

3.2 Related Work

On a high level, dynamic analysis based techniques use scenarios to trigger features of the system, gather execution traces while features are running, and later analyze those traces to suggest feature locations. Below, we describe how these techniques typically operate. Figure 3.2 shows a simple code snippet that can calculate the area of an isosceles triangle, an equilateral triangle or a rectangle based on the 'type' attribute of the given input. This system is invoked via a graphical user interface where the user selects the type of shape, inputs the attributes of the shape (e.g. width and height of a rectangle) and presses a button to get the answer from the system. When a developer needs to locate code that is highly relevant to 'calculating the area of an isosceles triangle', she creates three scenarios:

1. A scenario that will trigger calculating the area of an isosceles triangle. This is a positive scenario, where the feature of interest is invoked, denoted by t_1 .
2. A scenario that will trigger calculating the area of an equilateral triangle. This is a negative scenario, where a feature that we are not interested in is invoked, denoted by t_2 .
3. A scenario that will trigger calculating the area of a rectangle. This is again a negative scenario, denoted by t_3 .

Figure 3.3 shows, step by step, how the lines in the given code snippet are marked with the lines triggered when each scenario is executed. Step (a) shows the original code snippet from Figure 3.2. In step (b), after t_1 is executed, the set of triggered lines is:

$$L_{t_1} = \{1, 3, 4, 6, 10, 13, 25\}$$

```

1 read(type);
2 if (type = "triangle") {
3     read(a,b,c);
4     class := None;
5     if a=b or b=c
6         class := isosceles;
7     if a*a=b*b+c*c
8         class := right;
9     if a=b and b=c
10        class := equilateral;
11    case class of
12        right      : area := b*c/2;
13        equilateral : area := a*a*sqrt(3)/4;
14        otherwise  : s := (a+b+c)/2;
15                   area := sqrt(s*(s-a)*(s-b)*(s-c));
16    end
17 } else {
18     read(w,h);
19     if w=h
20         class := square;
21     else
22         class := rectangle;
23     area := w*h
24 }
25 write(class, area);

```

Figure 3.2. A simple code snippet that can calculate the area of an isosceles triangle, an equilateral triangle or a rectangle (example from Wong et al. [152]).

<pre> 1 read(type); 2 if (type = "triangle") { 3 read(a,b,c); 4 class := None; 5 if a=b or b=c 6 class := isosceles; 7 if a=a=b*b+c*c 8 class := right; 9 if a=b and b=c 10 class := equilateral; 11 case class of 12 right : area := b*c/2; 13 equilateral : area := a*a*sqrt(3)/4; 14 otherwise : s := (a+b+c)/2; 15 area := sqrt(s*(s-a)*(s-b)*(s-c)); 16 end 17 } else { 18 read(w,h); 19 if w=h 20 class := square; 21 else 22 class := rectangle; 23 area := w*h 24 } 25 write(class, area); </pre>	(a)	<pre> t1 1 read(type); t1 2 if (type = "triangle") { t1 3 read(a,b,c); t1 4 class := None; t1 5 if a=b or b=c t1 6 class := isosceles; t1 7 if a=a=b*b+c*c t1 8 class := right; t1 9 if a=b and b=c t1 10 class := equilateral; t1 11 case class of t1 12 right : area := b*c/2; t1 13 equilateral : area := a*a*sqrt(3)/4; t1 14 otherwise : s := (a+b+c)/2; t1 15 area := sqrt(s*(s-a)*(s-b)*(s-c)); t1 16 end t1 17 } else { t1 18 read(w,h); t1 19 if w=h t1 20 class := square; t1 21 else t1 22 class := rectangle; t1 23 area := w*h t1 24 } t1 25 write(class, area); </pre>	(b)
<pre> t2 t1 1 read(type); t2 t1 2 if (type = "triangle") { t2 t1 3 read(a,b,c); t2 t1 4 class := None; t2 t1 5 if a=b or b=c t2 t1 6 class := isosceles; t2 t1 7 if a=a=b*b+c*c t2 t1 8 class := right; t2 t1 9 if a=b and b=c t1 10 class := equilateral; t1 11 case class of t1 12 right : area := b*c/2; t1 13 equilateral : area := a*a*sqrt(3)/4; t2 14 otherwise : s := (a+b+c)/2; t2 15 area := sqrt(s*(s-a)*(s-b)*(s-c)); t2 16 end t2 17 } else { t2 18 read(w,h); t2 19 if w=h t2 20 class := square; t2 21 else t2 22 class := rectangle; t2 23 area := w*h </pre>	(c)	<pre> t3 t2 t1 1 read(type); t3 t2 t1 2 if (type = "triangle") { t3 t2 t1 3 read(a,b,c); t3 t2 t1 4 class := None; t3 t2 t1 5 if a=b or b=c t3 t2 t1 6 class := isosceles; t3 t2 t1 7 if a=a=b*b+c*c t3 t2 t1 8 class := right; t3 t2 t1 9 if a=b and b=c t3 t2 t1 10 class := equilateral; t3 t2 t1 11 case class of t3 t2 t1 12 right : area := b*c/2; t3 t2 t1 13 equilateral : area := a*a*sqrt(3)/4; t3 t2 t1 14 otherwise : s := (a+b+c)/2; t3 t2 t1 15 area := sqrt(s*(s-a)*(s-b)*(s-c)); t3 t2 t1 16 end t3 t2 t1 17 } else { t3 t2 t1 18 read(w,h); t3 t2 t1 19 if w=h t3 t2 t1 20 class := square; t3 t2 t1 21 else t3 t2 t1 22 class := rectangle; t3 t2 t1 23 area := w*h </pre>	(d)

Figure 3.3. The lines triggered by each scenario: (a) The original code snippet shown in Figure 3.2 (from Wong et al. [152]), (b) The lines triggered when scenario t_1 is executed, (b) The lines triggered when scenario t_2 is executed, (d) The lines triggered when scenario t_3 is executed.

Since this is a positive scenario, these lines are likely related to the feature we are interested in. In step (c), t_2 is executed and the set of lines triggered by this scenario is:

$$L_{t_2} = \{1, 3, 4, 6, 14, 15, 25\}$$

Finally in step (d), executing t_3 , the set of triggered lines is:

$$L_{t_3} = \{1, 18, 22, 23, 25\}$$

Since t_1 is the only positive scenario, and t_2 and t_3 are negative scenarios, the lines that were triggered by t_2 and t_3 are pruned from the set of lines triggered by t_1 :

$$\begin{aligned} L_{t_1} \setminus (L_{t_2} \cup L_{t_3}) &= \{1, 3, 4, 6, 10, 13, 25\} \setminus \{1, 3, 4, 6, 14, 18, 15, 22, 23, 25\} \\ &= \{10, 13\} \end{aligned}$$

In the end, a developer can start with the lines 10 and 13 in Figure 3.2 as locations in source code that are highly related to the feature of finding the area of an isosceles triangle.

Based on this description of how dynamic analysis based techniques typically operate, we review the existing literature below.

Software Reconnaissance [148] is the pioneering work on feature location. It uses a set of scenarios that execute a feature, and a set of scenarios that do not (negative scenarios) as described above. It then uses set-difference: execution units for a feature observed in the positive scenario are pruned by those observed in negative scenarios. Remaining execution units are the ones that are highly relevant to the feature.

Wong et al. [152] extended [148] with execution slices, which can use and suggest more types of execution units, such as branches and variables. Furthermore, they find out

the distribution of the implementation of a feature onto different components in source code (such as classes and packages), as well as how much each component participates in the implementation of each feature.

Antoniol and Gueheneuc [19] also extended [148] with statistical hypothesis testing, based on events that are observed in execution traces, knowledge-based filtering, and multi-threading support.

Eisenbarth's interactive technique [61] uses formal concept analysis, a mathematically sound technique to analyze binary relations, to assess the relationships between execution units and features. This technique provides information about unique execution units for each feature, which ones are shared and which ones are of interest to the implementation of a feature. In this regard, it provides both similar results as [152] and also additional information on relationships between features. This technique combines static and dynamic analysis to increase coverage of the dynamic information obtained, and it is interactive, i.e. it relies on the user to guide the analysis process and to investigate different types of relationships suggested by Eisenbarth's tool to locate code specific to each feature.

Poshyvanyk et al. [119] combined static and dynamic analysis to perform feature location. Instead of performing binary set operations as in [61], they use probabilistic ranking on the execution units observed in the execution traces of each feature. If a certain execution unit is observed in all scenarios for a feature, they propose that there is a high probability that this execution unit is highly relevant to the feature. They also use information retrieval to index textual information in the source code to help rank the relevant execution units discovered with dynamic analysis.

Liu et al. [97] introduced an interactive approach combining static and dynamic analysis. Their approach is similar to [119] in that they use dynamic analysis to find relevant locations for features first. Then, they ask users to perform queries relevant to

the feature they are interested in to rank the locations found via dynamic analysis.

Edwards et al. [59] addressed feature location on distributed systems. Distributed systems typically consist of multiple machines and they run continuously (they can't be stopped and started to gather execution traces for each feature), so it is hard to apply the previously discussed approaches to precisely know the start/stop times of when a feature is invoked. Furthermore, even though the start and stop times are known, synchronizing time across the distributed systems precisely is also difficult. Some distributed systems are event driven, which makes it harder to get the exact sequence of operations across the different runs of the same scenario. All these make distributed systems harder to apply the feature location techniques discussed above. Edwards et al. [59] tackle these problems by asking developers to mark start and end events of the execution of a scenario manually every time the scenario is executed. The scenarios are executed several times to remove noise (due to the asynchronous nature of distributed systems). Causality analysis is used to reorder events to deal with asynchrony. Finally, instead of using a set difference approach for positive and negative scenarios, they use component relevance indexes, a metric that tells how likely a component is related to the feature.

A different family of approaches builds on the feature location techniques described above to analyze the features themselves. These do not aim to solve the feature location problem, but rather use those techniques to process different properties of the features. Since they build on feature location techniques, we include them here for completeness. These techniques analyze feature relationships [60, 130], evolution of features across versions of software [71] and identifying canonical features of a system [85].

3.3 Limitations of Existing Techniques

In this section, we define the limitations of existing techniques, using our Chat System example described in Section 2.1.

Before we start the discussion, we first define feature dependencies, an important property between features that plays a key role in the success of feature location techniques.

Definition 3.3.1 (Feature Dependency). A feature f is said to be dependent on another feature f' , if f cannot be executed unless f' is executed first.

In our Chat System, a user cannot send a message unless she signs on to the server first. So `send-message` is said to depend on `sign-on`.

The input to feature location techniques that use dynamic analysis is typically the set of scenarios and their execution traces, which contain execution units.

Table 2.2 lists features and scenarios in our Chat System example, which are exercised manually on the system through a GUI while the system is being profiled.

First, as discussed in the previous section, dynamic analysis techniques depend on running scenarios for each feature. In our Chat System, the scenarios are executed manually using a GUI provided with the system. This puts a great burden on developers, especially if they need to perform feature location repeatedly, which is expected since the system can change due to bug fixes or new feature requests. Furthermore, if these actions are performed on the GUI manually as in our example, the collected execution traces will contain execution units that are valid at the time these scenarios are executed. If the code is refactored (e.g. names of classes/methods are changed), the execution traces will no longer be valid, and scenarios will need to be executed again. This is cumbersome for developers, therefore it is vital to automate this process.

```

@Scenario(feature="connect")
public static void scenario1() {
    // code to exercise "connect"
}
@Scenario(feature="sign-on")
public static void scenario2() {
    // code to exercise "connect"
    // code to exercise "sign-on"
}
@Scenario(feature="send-message")
public static void scenario3() {
    // code to exercise "connect"
    // code to exercise "sign-on"
    // code to exercise "send-message"
}
@Scenario(feature="sign-off")
public static void scenario4() {
    // code to exercise "connect"
    // code to exercise "sign-on"
    // code to exercise "sign-off"
}

```

Figure 3.4. Automation of feature scenarios. Scenarios in Table 2.2 are automated as small programs.

To automate this, one can implement small programs as scenarios for each feature, instead of manually executing the scenarios on the system. This is demonstrated in Figure 3.4. Each scenario contains code to exercise the respective feature annotated with the annotation *@Scenario* (this is meta-information on the program and is not executed). A user will need to implement these small programs to be used as scenarios, a cumbersome, labor-intensive and error prone process. Furthermore, this is not a simple task for especially new developers who joined a team recently. Some techniques require negative scenarios, which puts even more burden on users. Once implemented, the scenarios can be executed by a tool and their execution traces can be collected automatically. Note, however, that if the user is using an interactive technique (one that needs feedback from the user as it processes the execution traces), then the process of locating features will not be fully automated even though scenarios are captured as programs as in Figure 3.4.

Finally, some scenarios need to execute features other than the targeted feature due to feature dependencies. For instance, in Figure 3.4, the program for `send-message` needs to also execute `connect` and `sign-on` because it depends on both features. Almost all techniques in the literature require that scenarios are as simple as possible and invoke a specific feature and no others. This is required to avoid noise in the analysis of execution traces and provide relevant feature locations for each feature. Therefore, a user will need to preprocess the execution traces of these scenarios, most likely manually, to clean them up before applying a feature location technique. This is, again, a cumbersome, labor-intensive and error-prone process. These assumptions are common to almost all dynamic analysis based methods in the literature.

3.4 Contribution

Although recent techniques are known to work well, as discussed in the previous section, we observe that they have shortcomings on the assumptions they make. First, the

requirement to have distinct scenarios for each feature puts a great burden on developers. Even though developers can potentially use existing test cases as starting points and distill them into scenarios for features, this is still a manual task that is not repeatable reliably every time feature location needs to be performed. This is especially a bigger issue for users that are not very familiar with the system. Furthermore, negative scenarios are typically needed for some of the existing techniques, which puts more burden on users.

Second, if a process that automates the execution of the scenarios is not provided, it is very hard to keep consistent information to perform feature location reliably as software evolves. If execution traces are collected while scenarios are executed manually (e.g. on a graphical user interface of the system), feature location will not be repeatable as software is updated (e.g. code is refactored such that classes/methods are renamed), unless they are executed again on the updated system. Therefore, it is vital to have an automated process to execute scenarios without user intervention.

Finally, dependencies between features require users to do extra cleanup before execution traces can be used as input to tools. Many techniques assume that a given part of the execution trace will be related to only a specific feature and no others. In the presence of dependencies between two features, the dependent feature's execution trace will inherently contain all of the depended feature's execution trace. Therefore, users will typically need to clean up the execution traces for such dependent features to get good feature location results. This is a cumbersome and error prone process for users. Furthermore, dependencies between features also make it hard (and sometimes impossible) to create scenarios for the dependent feature, since the scenario for the dependent feature should ideally only trigger that feature and no others.

To the best of our knowledge, the work of Eisenbarth et al. [61] is the only one where use of scenarios that exercise multiple features is suggested and discussed. To

find feature locations specific to a feature, they look for execution units that only appear in the scenarios executed for a feature and nowhere else. This is a result of the nature of formal concept analysis (discussed later in this dissertation), since it makes binary decisions about the relationships between an execution unit and a feature. In the presence of dependencies between features, this technique may not find any feature locations specific to a feature at all, and require users to provide scenarios that exercise only that specific feature, i.e. require the manual cleanup discussed in the beginning of this section. Furthermore, this technique is interactive, i.e. it requires feedback from users during operation where users need to analyze a lattice produced by this technique to identify the feature location suggestions [97].

In this chapter, we bridge the shortcomings of the existing techniques discussed above by considering the feature location problem as a data mining problem. Given a set of test cases (in the existing test suite of the system) labeled with the features they exercise, we automatically find relevant feature locations in source code using association rule learning [13]. Our work makes the following contributions:

- We present a new way to find source code locations uniquely related to a feature, where users simply label some existing test cases from the system's existing test suite. A test case can exercise multiple features, and a feature may be tested in multiple test cases (a many-to-many relationship).
- We present a metric to guide users on the labelings they provide. If a feature cannot be located in a sufficiently reliable way, users are notified so that they can provide more labelings for that feature. This prevents users from labeling test cases blindly, without information on whether they provided enough input or not.
- We provide tool support to automate the entire process so that feature location can be performed reliably and repeatedly in the presence of feature dependencies even

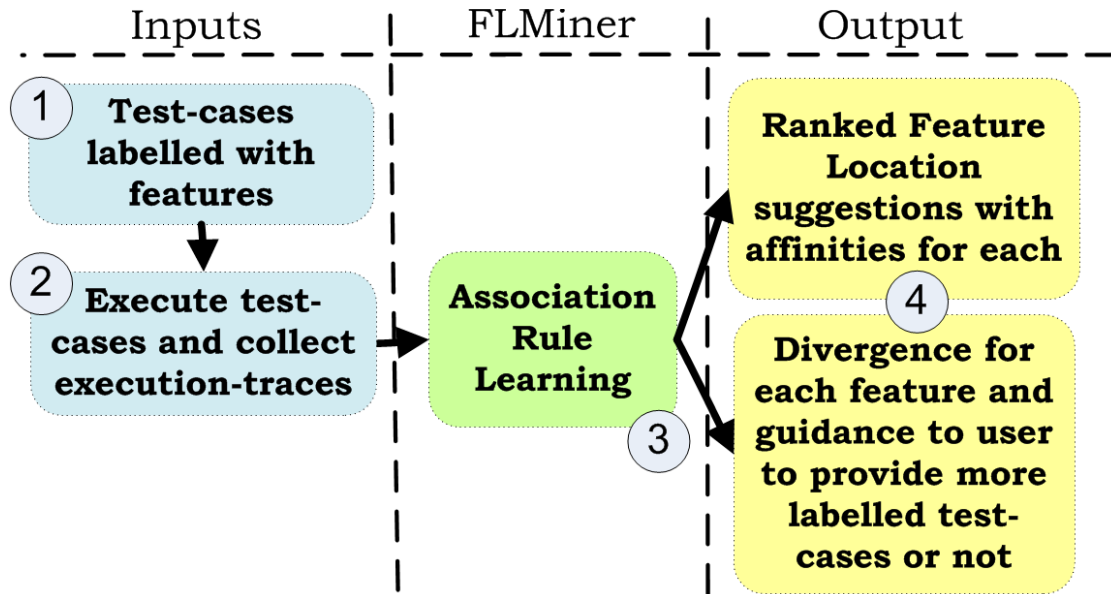


Figure 3.5. Overview of FLMINER, with inputs, its process and outputs. The inputs are the test-cases labeled with the features they execute. The outputs are the highly relevant feature location suggestions and feedback on the quality of the suggestions.

though software is updated.

3.5 FLMINER: Feature Location Miner

In this section, we describe our technique in detail. Figure 3.5 shows an overview of our technique with inputs and outputs. We implemented a tool for this process: FLMINER —Feature Location Miner. FLMINER takes in execution traces of test cases, uses association rule learning [13] to suggest highly relevant feature locations uniquely related to each feature, and finally outputs the found feature locations and some metrics on the quality of the found results as well as guidance for the user to provide more labeled test cases for higher quality results. We discuss each step in detail in the rest of this section.


```

@Scenario(features={"connect"})
public void testConnect() {
    // code to exercise "connect"...
}
@Scenario(features = {"connect","sign-on"})
public void testConnectAndSignOn() {
    // code to exercise "connect" and
    // "sign-on" in an unknown order...
}
@Scenario(
features={"connect","sign-on","send-message"})
public void testFeature6() {
    // code to execute "connect", "sign-on"
    // and "send-message" in an unknown order...
}
@Scenario(
features={"connect","sign-on","sign-off"})
public void testSignOff() {
    // code to execute "connect", "sign-on"
    // and "sign-off" in an unknown order...
}

```

Figure 3.6. Sample input to FLMINER for the Chat System.

3.5.1 Input to FLMINER

The input to FLMINER (Step 1 in Figure 3.5) is a set of test cases labeled with the features they execute. These are used as the scenarios for features. Figure 3.6 shows a sample input to FLMINER for our Chat System example. These test cases are executed, and execution traces are collected for each test case (Step 2 in Figure 3.5).

Compared to the sample scenarios in Figure 3.4, there are some important differences in the scenarios in Figure 3.6. First, the scenarios in Figure 3.4 are either created manually, or distilled from existing test cases in order to have as little noise as possible. Even though there are freely available tools for profiling and execution trace collection, users may still need to do manual cleanup on the execution traces due to dependencies, as discussed in the previous section.

Unlike the existing techniques, for FLMINER, users label existing test cases in

the test suite of the system with the features they execute, as in Figure 3.6. If they exist, we make use of the available execution traces collected by continuous integration tools after the execution of these scenarios. If not, we provide users with an AspectJ [5] aspect to output dynamic profiling information. Users do not need to do cleanup after providing information on which features are executed by which test cases.

Note that, as shown in Figure 3.6, scenarios can have multiple features specified in their labels. The execution traces collected by running such scenarios will contain a mixed collection of execution units, since the labels neither provide information on the order of execution of the features nor they provide information on which execution unit was executed due to which feature. This yields to, what we call, fuzziness.

Definition 3.5.1 (Fuzziness). As noted in [60], locating features, with the input in Figure 3.6, using existing techniques is a harder problem, because we do not have information on the order of the features executed in the test cases. Furthermore, when the test cases are executed and execution traces are collected, we do not know which part of the execution trace belongs to which feature. Therefore, there is inherent ambiguity in the information used by FLMINER. We call this ambiguity fuzziness.

3.5.2 Association Rule Learning and Confidence

Once the execution traces are collected, FLMINER uses a data mining algorithm, namely association rule learning [13], to find highly relevant locations for each feature (Step 3 in Figure 3.5). Association rule learning [13] is a popular method in data mining used to discover interesting relations between variables in a database.

Formally, association rule learning works on a set:

$$I = \{i_1, i_2, i_3, \dots, i_n\} \quad (3.1)$$

of n binary attributes, called *items*; and a set:

$$D = \{t_1, t_2, t_3, \dots, t_m\} \quad (3.2)$$

of m transactions, called the *database*. Each transaction in D contains a subset of the items in I .

A *rule* is defined as an implication $X \Rightarrow Y$, where $X, Y \subseteq I$ and $X \cap Y = \emptyset$. Intuitively, a rule corresponds to an implication of positive correlation between two items.

We demonstrate these concepts on the Chat System. Assume that we execute the test cases shown in Figure 3.6 and obtain information on which methods are executed in each test case, as shown in Table 3.2. For our example, the items are the union of all methods observed in the execution traces and the features; and the transactions are the test cases. In the table, a 1 denotes the existence of a method or feature in a test case. As an example, for the test `testConnectAndSignOn`, the execution trace contains the methods m_1 , m_2 , m_3 and m_4 ; and, as labeled by the user in Figure 3.6, the features `connect` and `sign-on` are executed by the test case.

Table 3.2. The items and database based on execution traces for the test cases in Figure 3.6.

Database Transactions	Items									
	m_1	m_2	m_3	m_4	m_5	m_6	connect	sign-on	send-message	sign-off
testConnect	1	1	0	0	0	0	1	0	0	0
testConnectAndSignOn	1	1	1	1	0	0	1	1	0	0
testSendMessage	1	1	1	1	1	0	1	1	1	0
testSignOff	1	1	1	1	0	1	1	1	0	1

In association rule learning, many different measures of interest and significance are defined [13]. In this chapter, we use *confidence*, since it captures the semantics of what our technique wants to achieve. Confidence is defined in terms of the *support* of a set of items:

$$\text{support}(X) = \frac{\# \text{ transactions with all items in } X}{\# \text{ total transactions}} \quad (3.3)$$

As an example, in Table 3.2:

$$\begin{aligned} \text{support}(m_3) &= \frac{3}{4} = 0.75 \\ \text{support}(\text{sign-on}) &= \frac{3}{4} = 0.75 \\ \text{support}(m_3 \cup \text{sign-on}) &= \frac{3}{4} = 0.75 \end{aligned}$$

Intuitively, support is the amount of information we have available for the given set of items, based on the transactions we have available. Next, the *confidence* of a rule is defined as:

$$\text{conf}(X \Rightarrow Y) = \frac{\text{support}(X \cup Y)}{\text{support}(X)} \quad (3.4)$$

As an example, in Table 3.2:

$$\text{conf}(m_3 \Rightarrow \text{sign-on}) = \frac{\text{support}(m_3 \cup \text{sign-on})}{\text{support}(m_3)} = \frac{0.75}{0.75} = 1$$

Intuitively, confidence is the extent of correlation between m_3 and `sign-on`, i.e. the likelihood that `sign-on` was executed as part of a test case given m_3 was observed in the execution trace of the test case.

For a method m to be considered as a highly relevant location for a feature f , the confidence $\text{conf}(m \Rightarrow f)$ should be high, since this would imply a strong correlation

between m and f . The confidence values for the rules between methods and features as described above is used, in part, by FLMINER to rank each method in deciding whether it is a highly relevant location for a feature or not. In our case studies, we set the minimum confidence to 0.5 based on our investigations on the outputs of our case studies, as it gave us the best results. For a more in depth analysis of association rule learning, we refer the reader to [13].

3.5.3 Divergence: Detecting Edge Cases

If we only use confidence to suggest locations for features, an edge case for our technique is a single test case that executes all features. In such a case, all confidence values of the form $conf(m \Rightarrow f)$ would turn out as 1. Obviously, this information is not helpful in choosing the highly relevant locations for features; because the confidence values turned out to be equal due to insufficient information input to FLMINER. In fact, as the input to FLMINER has more fuzziness (more labels on a single test case), the confidence values found by FLMINER are expected to be less reliable. Based on this observation, we present a new measure to perform an internal quality check on the confidence values found and counteract this edge case.

Consider a feature f and the set M of all methods such that:

$$M(f) = \{m \mid conf(m \Rightarrow f) > 0\} \quad (3.5)$$

We consider the confidence values of all of the methods in $M(f)$ as a probability distribution P_f :

$$P_f(m) = \frac{conf(m \Rightarrow f)}{\sum_{m' \in M(f)} conf(m' \Rightarrow f)} \quad (3.6)$$

and find out how much this distribution is different than a uniform distribution. If the distance is small, as was the case for the worst-case scenario described above (the distance

is 0 for that case, because all methods have confidence 1, and therefore $P_{\mathbf{f}}$ is a uniform distribution), then the confidence values are not very reliable. We call this distance *divergence*, and use KL-divergence [87], a well-known measure to calculate the distance between two probability distributions, to calculate it:

$$div(\mathbf{f}) = \sum_{m \in M(\mathbf{f})} P_{\mathbf{f}}(m) \ln \frac{P_{\mathbf{f}}(m)}{U(m)} \quad (3.7)$$

where $P_{\mathbf{f}}$ is the probability distribution constructed using the confidence values of methods in $M(\mathbf{f})$ as described above, and U is the discrete uniform distribution of the same size as $P_{\mathbf{f}}$. Note that, the divergence value is the same for all methods in $M(\mathbf{f})$ for a given feature \mathbf{f} . It can be considered as an indicator of the quality of the feature locations for \mathbf{f} found by FLMINER, and how much FLMINER considers them to be of high quality.

3.5.4 Affinity: Combining Confidence and Divergence

Finally, we combine confidence and divergence to obtain a single metric to be used to detect highly relevant feature locations. Given a method m and a feature \mathbf{f} :

$$affinity(m, \mathbf{f}) = 2 \times \frac{conf(m, \mathbf{f}) \times div(\mathbf{f})}{conf(m, \mathbf{f}) + div(\mathbf{f})} \quad (3.8)$$

Affinity combines confidence and divergence with equal weights, i.e. both confidence and divergence are equally important in determining whether a method is a highly relevant location for a feature. These weights can be adjusted based on project properties, to give more weight to either confidence or divergence after investigating the typical values they take for a project.

3.5.5 Outputs: Feature Locations, Affinity and User Guidance

Once affinities are determined for each (method, feature) pair, we rank the methods for a feature from high affinity to low affinity, and provide users with a list of the most highly relevant locations for each feature. FLMINER will also output the divergence values for each feature. In this problem, only providing more labeled input may not yield better results. Some choices of labeled test cases will sample features more evenly and provide better results. To help users on choosing test cases to label, FLMINER guides users to provide more labeled test cases for those features with low divergence. This way, the affinity values of methods can be increased for those features and the user can be presented with locations that are considered to be of higher quality, i.e. more relevant to a feature. When users follow FLMINER's suggestions, FLMINER yields higher quality results. The user will then know whether or not to label more test cases, and if so, for which feature (instead of blindly labeling test cases without any information on the quality of the results presented).

For our Chat System example, Table 3.3 lists the affinity values calculated for each method, and the divergence values for each feature. The divergence values highlight important information. For `connect`, divergence is 0, because the confidence values for all methods are equal for `connect`. So, FLMINER outputs that the given input was insufficient to find good feature locations for `connect`. Therefore, the user needs to provide more input for `connect` in order to provide better results. Furthermore, the divergence value for `sign-on` is lower than those for `send-message` and `sign-off`. Therefore, if the user prefers to provide more labeled test cases as input, `sign-on` would also be a good candidate.

Affinity values present to the user which locations in source code are the highly relevant ones for each feature. For `send-message`, m_5 is ranked highest (it has the

Table 3.3. Divergence values for each feature and affinity values for each (method, feature) pair. Highlighted cells (in bold and blue color) show which methods would be chosen as feature markers for each feature.

	connect	sign-on	send-message	sign-off
divergence	0	0.0086	0.1783	0.1783
affinity	connect	sign-on	send-message	sign-off
m_1	0	0.0169	0.2081	0.2081
m_2	0	0.0169	0.2081	0.2081
m_3	0	0.0170	0.2323	0.2323
m_4	0	0.0170	0.2323	0.2323
m_5	0	0.0170	0.3026	0
m_6	0	0.0170	0	0.3026

highest affinity), because it is only observed in the test case `testSendMessage` where `send-message` is present, while it does not exist in the other execution traces of the features `connect` and `sign-on`, which are also executed by `testSendMessage`. The divergence of `send-message` and `sign-off` are higher than that of `connect` and `sign-on`, because they have more distinguishing information available, i.e. `connect` and `sign-on` are executed in almost all test cases, while `send-message` and `sign-off` have been specifically executed in certain test cases. Therefore FLMINER presents them as better feature location suggestions.

3.6 Evaluation

To evaluate the effectiveness of FLMINER, we analyze it both theoretically and practically.

3.6.1 Baseline for Comparison: the Base Case

For comparison, we consider the same input that the existing techniques assume: a distinct scenario per feature. In such a case, the input would look like the one shown in Figure 3.4, where there is a separate scenario for each feature.

Furthermore, since most of the existing techniques assume that execution traces will be cleaned up so that each execution trace belongs to a single feature, we perform this operation too, i.e. we handle feature dependencies. We call this case the **base case**. Note that the base case is a very good input for a feature location technique since the execution trace for each scenario contains execution units for a single feature, which is known deterministically (the execution traces will also naturally contain utility methods and other unrelated methods for a feature, which is expected by all feature location techniques).

In the base case, the results found by FLMINER are identical to the well-known technique by Poshyvanyk [119]. In [119], conditional probabilities are used to calculate the probability of a certain method to be a relevant location for a feature (it combines this with static analysis as well, but we compare our method with its dynamic analysis part). The *confidence* measure we use as part of *affinity* also represents conditional probabilities. *Divergence* is the same for all methods for a feature, so it doesn't affect the feature location suggestions when used to calculate affinity for the base case. Therefore, FLMINER's analysis and results are equivalent to [119] when presented with the base case as input.

Measuring Success

We use the base case, i.e. the results of [119], as a baseline for our comparisons in the evaluation we perform. If FLMINER can achieve performance close to the base case when it is used on non-base case inputs, then we suggest that FLMINER will benefit users, since it allows simply labeling existing test cases and provides users guidance on which feature to provide more test cases for, while freeing developers off the burdens discussed in Section 3.3.

Below, we evaluate FLMINER for two cases: the ideal case and the practical case.

3.6.2 Ideal Case

Since FLMINER considers the feature location problem as a data mining one, it is expected that given 'enough' labeled inputs, it should be able to yield the same results as for the base case. How many labeled inputs can be considered as 'enough' depends on the nature of the input, i.e. the degree of fuzziness.

In the base case, there is no fuzziness: each scenario's execution trace contains execution units for one feature. As scenarios execute more features than 1, fuzziness increases. An example fuzzy input is shown in Figure 3.6, where some test cases execute multiple features.

Based on Figure 3.6, consider the case where the input is uniform in the number of features each test case executes: every test case in the input executes k features, where $1 \leq k \leq n$ and n is the number of features. In such a case, an ideal input to FLMINER would be $\binom{n}{k}$ examples, where the labels of the test cases are an enumeration of all of the k combinations of the n features. As an example, for $n = 3$ features: $\{f_1, f_2, f_3\}$ and $k = 2$, the ideal input to FLMINER would be three test cases, labeled with: (f_1, f_2) , (f_1, f_3) and (f_2, f_3) . If such an 'ideal' input is provided to FLMINER, we show below that FLMINER can find the same feature locations as the base case in spite of the fuzziness.

Our analysis starts with the confidence value of a method m for the feature f in the base case. Assume that, in the base case:

$$conf_{base}(m \Rightarrow f) = \frac{1}{1+x} \quad (3.9)$$

This means method m was observed in f 's execution trace and possibly some other features' as well (denoted by x , where $0 \leq x \leq n - 1$). To calculate $conf_{ideal}(m \Rightarrow f)$, we

find the following:

$$conf_{ideal}(m \Rightarrow f) = \frac{\# \text{ execution traces } m \text{ and } f \text{ are observed together}}{\# \text{ execution traces } m \text{ is observed}}$$

The number of execution traces m and f are observed together is:

$$\binom{n-1}{k-1}$$

since there are $\binom{n}{k}$ total execution traces, and we need to choose f as one of the features, and choose the rest $k-1$ from among the remaining $n-1$ features in the base case.

Number of execution traces m is observed is:

$$\binom{n}{k} - \binom{n-x-1}{k}$$

Note that m is observed in $x+1$ execution traces in the base case. To find the subsets it exists in the ideal case, we subtract the number of those execution traces where m isn't observed at all from the total number of subsets of size k :

$$\binom{n}{k} - \binom{n-(x+1)}{k} = \binom{n}{k} - \binom{n-x-1}{k}$$

Therefore, for the ideal fuzzy input to FLMINER, the confidence will be:

$$conf_{ideal}(m \Rightarrow f) = \frac{\binom{n-1}{k-1}}{\binom{n}{k} - \binom{n-x-1}{k}} \quad (3.10)$$

To analyze how the confidence values compare between the base and the ideal cases, we compare the ratio of the two confidence values as x changes. Since $0 \leq x \leq n-1$,

we analyze what happens as x converges to the boundary values.

$$\begin{aligned}
\lim_{x \rightarrow 0} \frac{\text{conf}_{ideal}(m \Rightarrow \mathbf{f})}{\text{conf}_{base}(m \Rightarrow \mathbf{f})} &= \lim_{x \rightarrow 0} \frac{\frac{\binom{n-1}{k-1}}{\binom{n}{k} - \binom{n-x-1}{k}}}{\frac{1}{1+x}} \\
&= \lim_{x \rightarrow 0} \frac{\frac{\binom{n-1}{k-1}}{\binom{n}{k} - \binom{n-1}{k}}}{\frac{1}{1}} \\
&= \lim_{x \rightarrow 0} \frac{\binom{n-1}{k-1}}{\binom{n}{k} - \binom{n-1}{k}} \\
&= \lim_{x \rightarrow 0} \frac{\binom{n-1}{k-1}}{\binom{n-1}{k-1}} \\
&= 1
\end{aligned}$$

$$\begin{aligned}
\lim_{x \rightarrow n-1} \frac{conf_{ideal}(m \Rightarrow \mathbf{f})}{conf_{base}(m \Rightarrow \mathbf{f})} &= \lim_{x \rightarrow n-1} \frac{\frac{\binom{n-1}{k-1}}{\binom{n}{k} - \binom{n-x-1}{k}}}{\frac{1}{1+x}} \\
&= \lim_{x \rightarrow n-1} \frac{\frac{\binom{n-1}{k-1}}{\binom{n}{k} - \binom{n-(n-1)-1}{k}}}{\frac{1}{1+(n-1)}} \\
&= \lim_{x \rightarrow n-1} \frac{\frac{\binom{n-1}{k-1}}{\binom{n}{k} - \binom{0}{k}}}{\frac{1}{n}} \\
&= \lim_{x \rightarrow n-1} \frac{\frac{\binom{n-1}{k-1}}{\binom{n}{k} - 0}}{\frac{1}{n}} \\
&= \lim_{x \rightarrow n-1} \frac{\binom{n-1}{k-1}}{\binom{n}{k}} \times n \\
&= \lim_{x \rightarrow n-1} \frac{\frac{(n-1)!}{(n-1-k+1)! \times (k-1)!}}{\frac{n!}{(n-k)! \times k!}} \times n \\
&= \lim_{x \rightarrow n-1} \frac{n \times (n-1)! \times (n-k)! \times k!}{(n-k)! \times (k-1)! \times n!} \\
&= \lim_{x \rightarrow n-1} \frac{n! \times (n-k)! \times k!}{n! \times (n-k)! \times (k-1)!} \\
&= k
\end{aligned}$$

To summarize:

$$\lim_{x \rightarrow 0} \frac{conf_{ideal}(m \Rightarrow \mathbf{f})}{conf_{base}(m \Rightarrow \mathbf{f})} = 1 \quad (3.11)$$

$$\lim_{x \rightarrow n-1} \frac{conf_{ideal}(m \Rightarrow \mathbf{f})}{conf_{base}(m \Rightarrow \mathbf{f})} = k \quad (3.12)$$

Therefore, the confidence values found in the ideal case will be a multiple (k) of the confidence values found in the base case for all methods m . When we calculate $affinity(m, \mathbf{f})$, we make use of $div(\mathbf{f})$ as well. Since divergence is calculated internally for each feature and is a linear transformation, it will not change the rankings found in the ideal case

(the affinity values) based on confidence values for the methods. As a result, in the ideal case, the methods will be ranked in the same order as they were ranked in the base case, and therefore FLMINER will find the same methods for each feature as relevant feature locations even in the presence of fuzziness.

Using the base and ideal case analysis above, we also analyze the result of changing the fuzziness, i.e. different values of k (noting that $1 \leq k \leq n$):

$$\begin{aligned}
 \lim_{k \rightarrow 1} \text{conf}_{ideal}(m \Rightarrow \mathbf{f}) &= \lim_{k \rightarrow 1} \frac{\binom{n-1}{k-1}}{\binom{n}{k} - \binom{n-x-1}{k}} \\
 &= \frac{\binom{n-1}{1-1}}{\binom{n}{1} - \binom{n-x-1}{1}} \\
 &= \frac{1}{n - (n-x-1)} \\
 &= \frac{1}{x+1}
 \end{aligned}$$

$$\begin{aligned}
 \lim_{k \rightarrow n} \text{conf}_{ideal}(m \Rightarrow \mathbf{f}) &= \lim_{k \rightarrow n} \frac{\binom{n-1}{k-1}}{\binom{n}{k} - \binom{n-x-1}{k}} \\
 &= \frac{\binom{n-1}{n-1}}{\binom{n}{n} - \binom{n-x-1}{n}} \\
 &= \frac{1}{1 - \binom{n-x-1}{n}} \\
 &= 1 \quad (\text{since } 0 \leq x \leq n-1)
 \end{aligned}$$

To summarize:

$$\lim_{k \rightarrow 1} \text{conf}_{ideal}(m \Rightarrow \mathbf{f}) = \frac{1}{1+x} \quad (3.13)$$

$$\lim_{k \rightarrow n} \text{conf}_{ideal}(m \Rightarrow \mathbf{f}) = 1 \quad (3.14)$$

Table 3.4. Information about case studies used to evaluate FLMINER

Case Study	# Features
UCSD Chat System	16
Apache Pool [3]	16
Apache Commons CLI [1]	11

Based on these, as fuzziness decreases ($k \rightarrow 1$), the confidence for a method found in the ideal case approaches the confidence found in the base case. As fuzziness increases ($k \rightarrow n$), the confidence of the methods approaches 1, regardless of their values in the base case. Therefore, as expected, having higher fuzziness makes it harder for FLMINER to suggest highly relevant feature locations.

This concludes our analysis that, given the ideal number of labeled test cases, FLMINER yields the same feature location results under fuzziness as the base case. Unfortunately, for a certain k , it would be impractical to expect users to label $\binom{n}{k}$ test cases. This could be a very large number of test cases to label, and there may not even be that many test cases in the test suite of the system. Therefore, the ideal case is not practical, but it is useful in showing the foundations of our technique.

In the next section, we evaluate FLMINER in a practical setting that can be realistically used by users.

3.6.3 Practical Case

In the practical case, we assess how many labeled test cases we can practically ask for from the users. Unlike the ideal case, it is not possible to mathematically calculate how FLMINER would behave since there is a wide range of possible inputs. Therefore, we performed regression experiments on three of our case studies: the UCSD Chat System, Apache Pool [3] and Apache Commons CLI [1]. Table 3.4 shows the relevant information for each case study used in our experiments.

First, the base case expects n test cases for n features. Since FLMINER is trying

to solve a more complex problem, i.e. it does not expect one scenario per feature, it is expected that it will need more test cases than n . An educated guess is that each feature should be observed at least twice to deal with fuzziness. Therefore, we tested to see how FLMINER would perform if $2n$ test cases are provided for n features.

Furthermore, as discussed in Section 3.5.5, if the users follow the guidance provided by FLMINER for each feature (through divergence), they would be expected to provide test cases labeled with those features pointed out by FLMINER. This is expected to keep the needed number of test cases to a minimum by guiding users to provide input where it is most needed for higher quality results, hence less work for users. Since users are expected to follow this guidance for high quality results, we used inputs that conform to this guidance in our experiments.

Experiment Setup

To perform experiments, we first identified features for each case study through their documentation. Next, we created scenarios conforming to the base case, i.e. a distinct scenario for each feature (including the cleanup needed due to feature dependencies). Then we executed the scenarios and collected execution traces for each feature using AspectJ [5].

Next, we generated random inputs of different labelings using the execution traces for features obtained above: some with one feature, some with two and so forth, up to six (we did not go beyond six because the maximum number of features in a test case across all case studies was five). As an example, to simulate a test case with two features, we simply combined the execution traces of two features from the base case and labeled the union with the two features. Note that, regardless of how the random input has been generated, the total number of test cases is fixed to $2n$ in each case study.

We generated the random inputs in two steps. First, we enumerated all possible

labeling counts that add up to $2n$, such that there is at least one sample scenario for each feature.

As an example, if a case study had three features (i.e. $n = 3$), we enumerated all solutions to the following linear equation:

$$x_1 + x_2 + x_3 = 6$$

Here, for a given solution to this equation, x_1 corresponds to the number of test cases where there is a single feature, x_2 with two features, and x_3 with three features. By enumerating all solutions, we ensure that there was no skew on how we sample the random space while we generate the random inputs. The possible enumerations solving the linear equation above are:

x_1	+	x_2	+	x_3	= 6
1	+	1	+	4	= 6
1	+	2	+	3	= 6
1	+	3	+	2	= 6
1	+	4	+	1	= 6
2	+	1	+	3	= 6
2	+	2	+	2	= 6
2	+	3	+	1	= 6
3	+	1	+	2	= 6
3	+	2	+	1	= 6
4	+	1	+	1	= 6

Then, for each given solution to the linear equation, we randomly chose features to assign to test cases, such that each feature is used at least once. We repeat this process 100 times so that for a single solution to the linear equation, there are 100 random inputs

to be used in our experiments. As an example, for the first enumeration shown above:

$$x_1 + x_2 + x_3 = 6$$

$$1 + 1 + 4 = 6$$

we generate 100 random inputs such that there is 1 test case with a single feature, 1 test case with two features and 4 test cases with three features. Also note that, the randomly generated labelings are filtered to actually make sense, where each feature is represented in at least one execution trace. This is possible, since there are n features, and $2n$ randomly generated execution traces.

Evaluation Criteria

To evaluate FLMINER’s performance on the random inputs, first, we describe what a ‘highly relevant’ feature location is. Based on the existing literature, these locations are typically uniquely related to a feature. Therefore, observing a highly relevant feature location in an execution trace implies that the scenario of that execution trace executed that feature.

Based on this observation, given the highest ranked feature location l suggested by FLMINER for a feature f , and the execution trace of a test case t , FLMINER is said to guess that t executed f if the execution trace of t contains l . If l is indeed a highly relevant feature location for f , then FLMINER’s guess is correct.

For each case study, we manually created a baseline to assess how well guesses of FLMINER are, by looking at each test case and finding out which features it actually executes.

Finally, we compared how well FLMINER’s guesses were compared to the

baseline, using the accuracy metric f-measure [141] introduced in Section 2.4.1. In this experiment, the relevant set is the set of trace links between features and test cases, while the retrieved set is the set of trace links found by using the highly relevant feature locations found by FLMINER. Given a guess g made by FLMINER, g is either correct, a false-positive (FLMINER guessed that a test case executed a feature, but it was wrong), or a false-negative (FLMINER missed the fact that a test case executed a feature).

Furthermore, we also performed the same experiments with base case inputs to FLMINER for each case study and reported the f-measure values for the base case. As discussed in Section 3.6.1, the base case for FLMINER is equivalent to Poshyvanyk’s method [119]. Therefore, we provide a comparison of FLMINER’s performance on fuzzy inputs with that of [119] on base case inputs.

Evaluation Results

Figure 3.7 shows the results of our experiments, where we use $2n$ test cases for n features as input to FLMINER. In the graphs, the line at the top for each case-study shows the f-measure obtained in the base case, which is equivalent to what would be obtained using the method in [119]. The scattered points show the f-measure values obtained by FLMINER on the random experiments. Linear fit for the scattered points are also shown, with information on the average f-measure value obtained by FLMINER for each case-study. As shown, FLMINER does not perform as good on fuzzy inputs as the base case. This is expected, since the fuzzy inputs are more complex than the base case. However, on average, FLMINER performs 83% —97% as good as the base case, i.e. Poshyvanyk’s method [119]. Furthermore, as discussed in Section 3.3, FLMINER does not require creating any scenarios and the preparation required on the execution traces (due to dependencies), unlike the existing feature location methods. It only needs some labeled test cases from the existing test suite of the system.

Table 3.5 shows the results of the same experiment when the number of labeled test cases is increased from $2n$ (as in Figure 3.7) to $3n$. When more labeled test cases are provided, FLMINER is expected to perform better because it has more information to learn from. As expected, using $3n$ examples increases the performance of FLMINER for all case-studies.

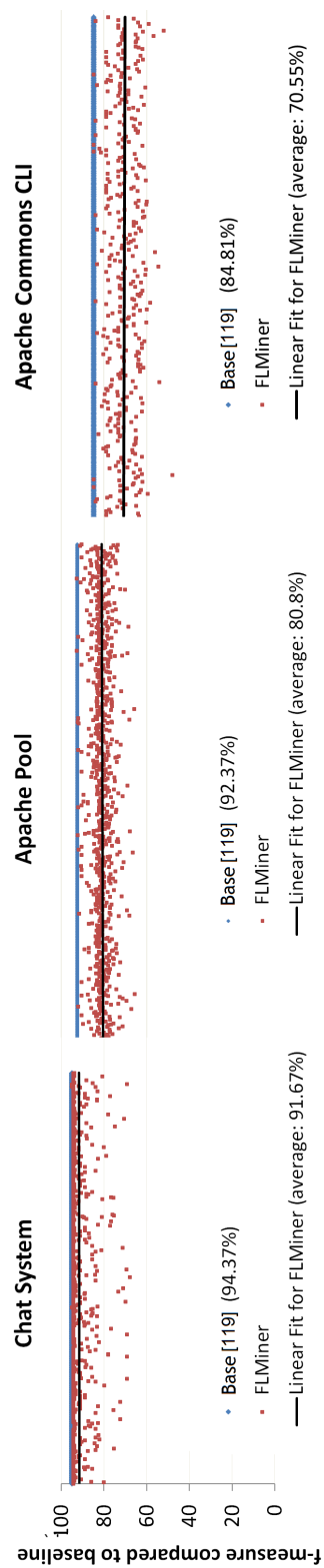


Figure 3.7. FLMINER performance with $2n$ fuzzy test cases as input (n is the number of features), compared to the base case input [119].

Table 3.5. Average f-measure based on the number of labeled test cases provided as input for each case study in the evaluation of FLMINER.

Case Study	Average f-measure	
	Number of Test Cases Provided	
	$2n$	$3n$
UCSD Chat System	91.67%	94.11%
Apache Pool [3]	80.80%	85.20%
Apache Commons CLI [1]	70.55%	77.05%

3.7 Discussion

In this section, we discuss the advantages and disadvantages of our technique.

FLMINER lifts the assumption of feature location techniques that each scenario should execute a single feature. We use existing test cases of the system instead of asking users to create scenarios. This decreases the burden on developers due to scenario creation and the cleanup needed due to feature dependencies.

In the evaluation section, we show that FLMINER can achieve results on fuzzy inputs close to one of the well-known existing techniques [119] on the base case.

FLMINER also provides guidance to users whether any more labeled test cases can be used to increase the quality of the results. FLMINER does not require too many test cases as input: we show that it performs well when it is given about $2n$ labeled test cases where there are n features, given that the developers follow the guidance provided by FLMINER. Furthermore, if more labeled examples are provided, its success improves further (see Table 3.5).

Our technique is independent of the programming language used in the system. The only requirement is the existence of a profiler. We implemented FLMINER for Java [68], however it can easily be extended to work for any language. Our technique uses test cases of the system. Therefore, it is not applicable to systems without test cases. However, it is common for production systems to have test cases available [107].

Therefore, FLMINER can be adopted in many production systems.

Another advantage of our method is that the test cases can be executed without manual intervention from users. This makes the task of locating features repeatable, and our process can be integrated into continuous integration tools for complete automation. This way, users don't need to carry out any actions: feature location suggestions will be ready every time continuous integration runs the test cases and feeds the input to FLMINER.

Similar to existing techniques in the literature, our technique only supports functional requirements. Pruning utility methods is commonly used in existing techniques to improve feature location performance. In FLMINER, we allow users to do this through the use of regular expressions.

Finally, our technique builds upon dynamic analysis and the use of scenarios (test cases). So it carries the same limitations as existing techniques about coverage, i.e. capturing different ways of executing a feature across the system. This can be mitigated by integrating a technique to improve coverage as described in the next chapter (discussed in Table 4.3) into FLMINER.

3.7.1 Threats to Validity

In this section, we discuss the issues that might have affected the results of our case studies presented in Section 3.6.3, and therefore may limit the interpretations and generalizations of our results.

First, we cannot claim that our case studies represent the full extent of production systems. We chose our case studies from different domains to mitigate this risk, and two of them are commonly used production software. This risk can be further mitigated if we experiment with more case studies from more domains.

Second, we are not domain experts of the software used in our case studies.

Therefore, we cannot claim that we found all features in each system, and that the scenarios we created are the best ones to capture them. Therefore, depending on the chosen features and scenarios, the results may differ.

Third, in our approach, as the number of features increases, the effort necessary to label test cases as training data also increases. Our approach needs to be evaluated on larger systems with higher number of features to assess how well it scales.

Finally, we created the baseline for the experiments in Section 3.6.3 manually. To mitigate risk, we had two different developers perform this task and compare their results. However, mistakes might still have happened.

3.8 Conclusion

In this chapter, we presented a dynamic analysis based feature location technique and a tool, FLMINER, that uses data mining on existing test cases of a system to suggest highly relevant feature locations uniquely related to features. Similar to existing dynamic analysis based feature location techniques, FLMINER makes use of scenarios. However, it has the following improvements over the existing techniques:

1. it doesn't require users to create scenarios, users can simply tag existing test cases with the features they execute
2. it doesn't require a distinct scenario for each feature, test cases can execute multiple features and a feature can be executed in multiple test cases
3. it doesn't require users to perform cleanup on the execution traces of scenarios due to feature dependencies.

FLMINER provides users guidance on the quality of the suggested feature locations and whether the results can be improved if more labeled test cases are provided for

a feature. Furthermore, it provides an end-to-end automated process to locate features from the execution of test cases automatically to outputting feature location suggestions.

On fuzzy inputs, where a scenario can execute multiple features and a feature might be executed by multiple scenarios, our technique yields results, on average, within 83% —97% of the results provided by a well-known successful technique [119] on base case inputs (a distinct scenario for each feature).

Based on experiments on three case studies, $2n$ labeled test cases yield the results described above, where there are n features. Furthermore, more labeled examples yield better results.

The work in this chapter, in full, is a reprint of the material as it appears in: “Celal Ziftci and Ingolf Krüger. Feature Location Using Data Mining on Existing Test-Cases. In the 19th Working Conference on Reverse Engineering, pages 155-164, Kingston, Ontario, Canada, 2012. IEEE.” The dissertation author was the primary investigator and author of this paper.

3.9 Future Work

As future work, FLMINER can be complemented with static analysis to yield better results, as in [61, 119, 97]. It can also be improved to work for non-functional requirements. Currently, we evaluate FLMINER against [119] by combining execution traces of scenarios for each feature. As future work, FLMINER should be evaluated on the whole test-suite of each case study. As part of this experiment, it is important to assess how fast it converges and stops for asking for more labeled test cases. Finally, the interaction of developers and FLMINER should be assessed on a user study to investigate the usefulness of FLMINER’s guidance on labeling test cases as input.

Chapter 4

Tracing Requirements to Tests with High Precision and Recall

In this chapter, we focus on tracing requirements in tests. We build upon the feature location techniques introduced in Chapter 3 to find highly relevant feature locations for requirements. Then we use those locations to find requirements traces in test cases.

4.1 Introduction

Testing is an important part of the software development lifecycle, has many benefits for the final product, and is employed by many, if not all, software development teams. Empirical evidence suggests that, in many software systems, the amount of test code ranges between 0.5 to 1.5 times the amount of code produced for the system itself [107, 149]. This ratio increases further for some systems, such as critical systems or systems that follow the Test Driven Development process [74].


Having many tests increases the effort spent on testing, its cost in the development process, and the importance of tracing requirements in tests. Requirements traces in tests demonstrate the customers which requirements are tested to achieve reasonable quality [135]. For product developers, testers and managers, they show which requirements are covered by which test cases. For new developers joining a team or customers using the

software, they provide a summary of the test cases to examine as usage examples and to help understand the software. For a testing lead, they help prioritize testing efforts [69].

These testing and software quality related activities are possible only if requirements traces are accurate, complete and up to date. There is tool support, such as DOORS [6], to record, manage and retrieve trace information manually. However, acquiring and maintaining accurate traces manually is an error-prone, time consuming, and labor-intensive process that requires disciplined developers [69, 34, 99]. Furthermore, the risk that RT links get out of date as software evolves is high due to this manual effort. Therefore, for both legacy and new systems, it is important and convenient to establish, maintain and retrieve requirements traceability links using an automated process.

Overall, tracing requirements in tests can be summarized as shown in Figure 4.1, where the aim is to start from a description of requirements and get to a set of requirements trace links in tests (an RTM in our case).

ID	Requirement Name	Requirement Description
1	connect	Users should connect to the server before they start doing anything else. Connect operation connects to the backend server and gets a listing of all users currently connected.
2	sign-on	After they connect, users should sign on to the backend server with their credentials. The backend server will return a success or failure message to the client upon attempt to sign on.
3	send-message	Once they sign on to the backend server, users can send messages to each other. Messages should have a timestamp so that they can be ordered by the backend server.
4	sign-off	Once users are done using the system, they can sign out of the backend server. This will end their session, and free up resources in the backend server. Only users that have signed on successfully can sign off.



t_1 : testConnect	connect	✓		
t_2 : testConnectAndSignOn		✓	✓	
t_3 : testSendMessage		✓	✓	✓
t_4 : testSignOff		✓	✓	✓

Figure 4.1. Tracing requirements in tests. We start with the descriptions of requirements (the requirements specification document), and the aim is to get to the requirements traces in tests, in the form of an RTM in our case.

4.2 Related Work

In this chapter, we use precision and recall (discussed in Section 2.4.1) to assess the quality of requirements tracing techniques. In this context, as shown in Figure 4.2, the relevant set is the set of requirements traces (the ticks) in the RTM between requirements and tests. The retrieved set is the set of requirements traces suggested by a technique. Precision, then, is the correctness of the requirements traces suggested by the technique, while recall is the completeness of the actual requirements traces with respect to the suggested traces by the technique.

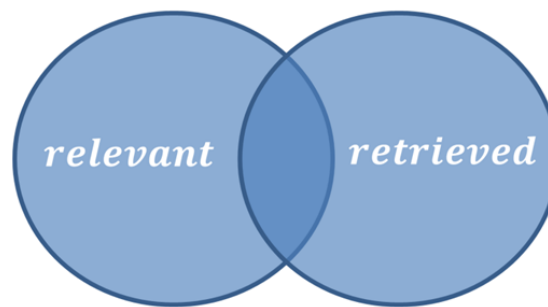
For RT purposes, it is important to obtain recall values close to 100%, because this represents finding all traceability links [156]. However, it is also important to obtain high precision in high recall ranges, since this corresponds to low number of false positives, and fewer trace links to be analyzed by humans manually to verify correctness [98, 156].

Different approaches emerged to automatically retrieve trace links to different types of artifacts, such as source code, tests, and design documents. We provide an overview of these approaches relevant to our work in Figure 4.3 and discuss each in more detail in the rest of this section.

A significant body of research on tracing requirements to source code already exists. There are approaches that use the documentation of requirements and source code of the system to perform IR and find traceability links [18, 100, 75, 98, 106] (see 2 in Figure 4.3). These approaches assume that same or similar terms are used in the documentation and the source code. They perform textual indexing on both documentation and source code and compare the terms in each index to find similarities. These approaches aim to find all traceability links, i.e. to obtain high recall. The advantage of these approaches is that, since they work with text, they can find traceability links not only in source code but also in other types of text-based artifacts (see 1 in

relevant: the actual requirements traces in tests

retrieved: the requirements traces found and suggested by a technique



$$precision = \frac{|relevant \cap retrieved|}{|retrieved|}$$

$$recall = \frac{|relevant \cap retrieved|}{|relevant|}$$

Figure 4.2. Precision and recall in the context of tracing requirements in tests. The relevant set is the set of requirements traces (the ticks) in the RTM between requirements and tests. The retrieved set is the set of requirements traces suggested by a technique. Precision, then, is the correctness of the requirements traces suggested by the technique, while recall is the completeness of the actual requirements traces with respect to the suggested traces by the technique.

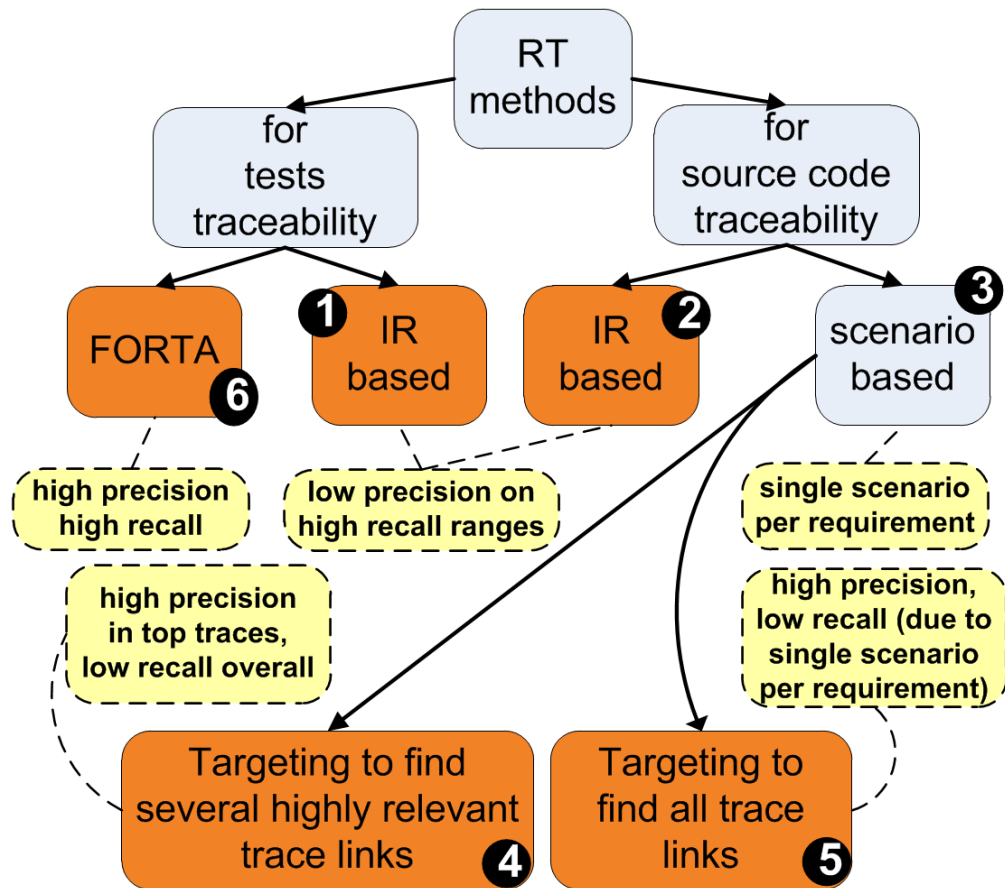


Figure 4.3. Classification of related work for FORTA.

Figure 4.3). The disadvantage of these approaches is that they need documentation for requirements, and proper naming conventions and comments in source code, i.e. well-documented software, to yield accurate results. Typically, they suffer from low precision (many false positives) on high recall values [98, 156].

A different family of approaches to tracing requirements in source code uses executable use-cases, called scenarios, to represent requirements [60, 148, 152, 151, 61], with some additionally using static source code analysis [19, 39] (see 3 in Figure 4.3). A scenario can be a test case, manual execution of the requirement on the software by a person, or any other specification that represents a requirement and can be converted into an executable form. These approaches propose executing these scenarios, gathering program execution traces as they run and analyzing the execution traces with different techniques (reconnaissance [148], execution slicing [152], formal concept analysis [61], probabilistic ranking [119], footprint graph [60] and many more) to find out where requirements are implemented in the source code, hence finding traceability links.

Some of these source code traceability approaches target finding several highly relevant requirements traces in source code but not finding all traces (high precision in top traces with low recall overall) [148, 152, 119] (see 4 in Figure 4.3). These highly relevant traces serve as starting points for developers to investigate the code-base further to gain an understanding of the requirement. These approaches achieve good precision in the requirements traces they retrieve, however they obtain low recall overall.

The rest of these source code traceability approaches target finding all trace links [60, 61] (see 5 in Figure 4.3). Similar to the other scenario based approaches, they assume using a single scenario per requirement.

Overall, one of the advantages of these scenario based source code traceability approaches is that they do not require any documentation; they only use executable scenarios. They also provide good accuracy, given that they analyze the requirements

on a running system with accurate specifications of requirements via scenarios [60]. However, unlike the IR methods, these approaches only apply to source code traceability. Furthermore, these approaches require executable scenarios specific to each requirement, which may not exist readily in some systems. Also, these approaches can only retrieve trace links for functional requirements of the system unless they are supported with manual effort during the analysis [60]. Finally, these approaches can only gather trace links on those parts of the system that are exercised by the scenarios, which may result in missing some of the trace links. Given that these approaches assume representing each requirement with a single scenario, some trace links are inevitably missed for some requirements.

There are approaches for RT in source code that combine IR methods with usage of scenarios and static analysis [19, 20], and increase the accuracy of the methods described above. To increase the precision of the IR approaches at high recall values, one study used term-based enhancement methods [156] and reported that although some improvements can be achieved, the results vary based on project properties such as documentation characteristics and domain specific vocabulary. Another study used static source code analysis for the same purpose, and reported minor improvements on top of the IR methods [109].

The focus of this chapter is retrieving traceability links in test code (see 6 in Figure 4.3) to overcome the challenges of the IR approaches [18, 100, 75, 98, 106] (see 1 in Figure 4.3) by building upon source code traceability methods. Even though some studies using the IR methods report that they observed slightly better test traceability than source code traceability [98], these approaches still suffer from low precision in high recall values as discussed above.

Our method builds on the use of scenarios to find the location of several highly relevant trace links for functional requirements in source code [119], and then uses them

to find requirements traces in tests. Unlike existing source code traceability techniques [60, 61], our approach allows using multiple scenarios for a single requirement, which yields higher recall. Compared to IR-based test-traceability approaches, our method uses a more accurate description of requirements, which results in accurate traces in source code, which in turn results in more accurate traces in tests, i.e. high precision.

4.3 Contribution

In this chapter, we use features to automatically create traceability links between requirements and test cases (including any type of executable tests such as unit, acceptance, system and integration tests). Our work makes the following contributions:

- A new method to find traceability links between functional requirements and any type of executable tests, building on the existing requirements to source traceability approaches. Our method improves upon precision/recall values obtainable by existing IR approaches, because it uses a more accurate, executable description of requirements. In our case studies, we observed precision/recall values higher than 90%, whereas the IR techniques obtain lower values.
- A method for increasing recall in obtaining requirements traces in tests by supporting multiple ways of triggering a requirement, where existing techniques that assume a single way of triggering a requirement obtain lower recall.
- An automated process and tool, FORTA, to create traceability links between requirements and tests as a by-product of automated software development processes, such as TDD and continuous integration. Unlike the existing IR approaches, the traceability links stay up to date, because the requirements are represented using executable specifications.

4.4 FORTA: Tracing Requirements to Tests Via Features

Figure 4.4 summarizes the inputs, flow and outputs of our approach and tool. We outline the steps here, and each step is explained in detail in the following sub-sections.

In our approach, the first step is identifying the features (Step 1 in Figure 4.4). This is typically done using the documentation of the system or talking to the stakeholders (details in Section 4.4.1).

Second, we create scenarios to exercise each feature (details in Section 4.4.1). As each scenario is executed, our tool gathers execution traces using a profiler or a similar technology. Execution traces contain execution units, such as class and method names (Step 2 in Figure 4.4).

Third, we use the execution traces for a scenario, and find execution units that distinguish features from each other, i.e. execution units that exist in that feature but no others (Step 3 in Figure 4.4). We name these distinguishing execution units feature markers (details in Section 4.4.2).

Next, we execute the tests of the system and again record their execution traces (Step 4 in Figure 4.4). These traces contain the same type of information with the execution traces collected earlier for scenarios.

Then, in the execution traces of the tests, we find the feature markers for each feature (Step 5 in Figure 4.4). This is a contribution of our method and reveals which test cases exercised which features (details in Section 4.4.3). This way, we find the traceability links between requirements and test cases and capture them in the form of a Requirements Traceability Matrix (Step 6 in Figure 4.4).

FORTA —Feature Oriented Requirements Traceability Analysis —is our tool that implements this process. The rest of this section explains the steps in Figure 4.4 in more

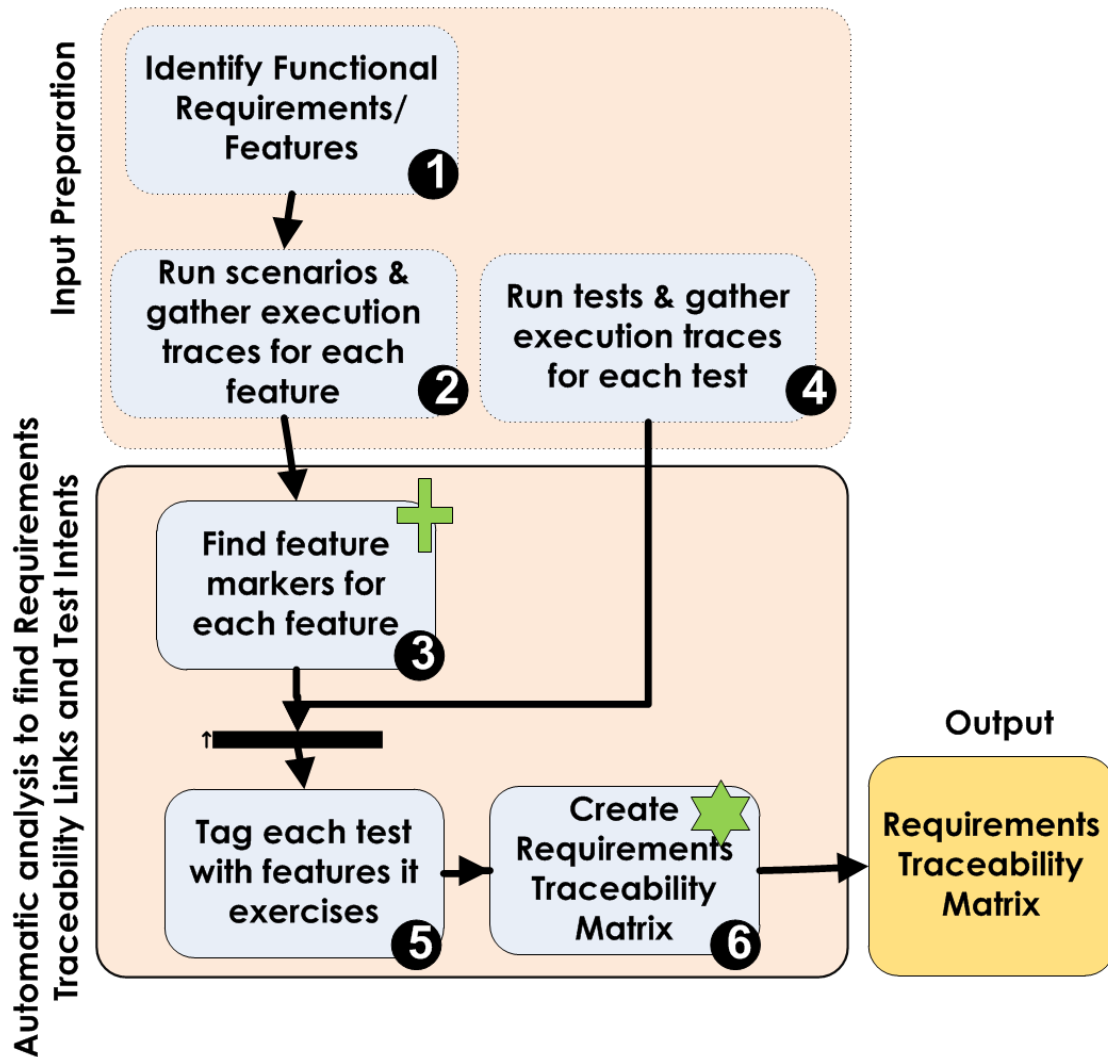


Figure 4.4. Inputs, outputs and flow of FORTA. The inputs are execution traces of scenarios and tests, while the output is the Requirements Traceability Matrix between requirements and tests. The steps with a * indicate a novel contribution of our approach, while steps with a + indicate that we use existing research and make a contribution additionally.

detail.

4.4.1 Features and Scenarios

To invoke a feature, one needs to trigger it by using scenarios. In our case, we performed the scenarios described in 2.2 for our Chat System using the provided graphical user interface.

Finding features is mostly a manual task, unless they are already specified in a requirements specification document. Preparing scenarios for features is similarly mostly manual, unless feature descriptions are accompanied with directly or indirectly executable scenarios as in tools like FIT [8]. These correspond to Step 1 in Figure 4.4.

4.4.2 Feature Markers

Once we have the scenarios for features from Step 1, we execute them on the system and collect execution traces for each feature using a profiler (Step 2 in Figure 4.4). These execution traces contain which execution units were executed while the scenarios were running (there can be multiple scenarios for a feature). The choice of execution units depends on two things: the programming language and the level of abstraction needed. Execution units can be procedure names along with file names each procedure is in for procedural languages; they can be class and method names for object oriented languages; and they can be namespaces and function names for functional languages. To obtain higher level information, the execution units can be chosen at a higher level, such as package name or namespace only.

Once we collect the execution traces for each feature, we find execution units that are specific to each feature (Step 3 in Figure 4.4). There is existing research to find specific locations in source code to investigate first to gain an understanding of a feature [148, 152, 61, 119]. These techniques are good at finding execution units that are

specifically related to a feature, providing good starting points for further investigation. We use one of these techniques known to perform well [119], to find distinguishing execution units among features which we call **feature markers**. As an example, consider the following features and the methods observed in their execution traces in the Chat System example:

$$\begin{aligned} \text{connect} &: (m_1, m_2, m_3) \\ \text{sign-on} &: (m_3, m_4) \\ \text{send-message} &: (m_1, m_3, m_5) \\ \text{sign-off} &: (m_3, m_6) \end{aligned}$$

To represent connect, we choose m_2 because that is the only method observed in its execution trace and no others. Similarly, we choose m_4 for sign-on, m_5 for send-message and m_6 for sign-off.

Obviously, we may not be able to find such distinguishing methods for every feature. So instead of finding the feature markers for each feature in a binary fashion as described above, we use probabilistic ranking as done in [119]. For the same example, the ranking is given as:

$$\begin{aligned} \text{connect} &: (m_1 : 0.5, m_2 : 1.0, m_3 : 0.25) \\ \text{sign-on} &: (m_3 : 0.25, m_4 : 1.0) \\ \text{send-message} &: (m_1 : 0.5, m_3 : 0.25, m_5 : 1.0) \\ \text{sign-off} &: (m_3 : 0.25, m_6 : 1.0) \end{aligned}$$

Since m_3 is observed in all four features, its ranking is $\frac{1}{4}$. All rankings for all observed methods are calculated similarly, and then they are ranked for each feature.

After this ranking, a constant number is used to pick some of the methods to be feature markers for each feature using their probabilistic rankings. For our Chat System example, if we choose the constant to be 1, the feature markers would be:

connect : $[m_2]$
 sign-on : $[m_4]$
 send-message : $[m_5]$
 sign-off : $[m_6]$

Note that, as done in [61], we also experimented with using formal concept analysis [28, 66] to choose the feature markers and we observed similar results.

Our approach takes existing research [61, 119] further by using multiple scenarios for a single feature, using scenarios separately to find feature markers for each of them, and then combining all markers to represent a single feature. This way, even though a feature might have multiple scenarios (i.e. different ways of being triggered), we get at least one execution unit for each scenario in the feature markers, and do not miss trace links due to competing scenarios for the same feature.

4.4.3 Tracing Features to Test Cases

This step is a contribution of our approach. To be able to collect traceability links in tests, we execute the test cases and collect execution traces while they are running using a profiler (Step 4 in Figure 4.4). The collected execution traces contain the same type of execution units described in the previous subsection for scenarios.

Given the feature markers for each feature, we look for the execution units in the feature markers inside the execution traces of each test (Step 5 in Figure 4.4). For an example, consider the feature marker for connect and the execution trace for the test

testConnect:

$$\begin{aligned} & \text{connect} : [m_2] \\ & \text{testConnect} : (m_2, m_4, m_7, m_{13}, m_{146}) \end{aligned}$$

Since testConnect's trace contains one of the feature markers of connect, we say that testConnect exercises connect, hence there is a traceability link between them. Doing this for each feature and test, we get an RTM as shown in Table 2.3 (Step 6 in Figure 4.4).

4.4.4 Tool Support

We provide automated tool support for our approach, which can be integrated into automated software processes such as continuous build and TDD. In a continuous build setup, our approach can be run as a task similar to running tests, packaging and providing code coverage. The input to our system are execution traces for each feature and execution traces for tests. Given these, our tool outputs a report with the RTM and the rankings of each feature for each test to show the real intent of the test.

Automated test cases are typically marked with special identifiers (such as *@Test* for JUnit [10] in Java [68]). We propose using automated tests as feature scenarios as well with an identifier (*@Scenario(requirement="")* in Java [68]). This way, continuous build systems can run the tests and scenarios, collect execution traces for both, and then feed them into FORTA automatically.

4.5 Evaluation

To assess the validity of our approach, we used all four case studies described in Section 2.3.

Table 4.1. FORTA case study properties.

Case Study	# Features	# Test Cases
Chat System	16	20
Apache Pool [3]	16	77
Apache Log4j [2]	10	69
Apache Commons CLI [1]	11	181

For comparison, we implemented two well-known, recent IR techniques used to trace requirements to tests: “Term Frequency Inverse Document Frequency” (TF-IDF) [81], and “Latent Semantic Indexing” (LSI) [51]. As input, they take the documentation for a requirement, and find terms in this document. They also take the source code of a test as text and find terms in it, too. Then they match the terms in the requirements document and in the test document to see how closely they match (based on the frequency of each term in each document) using a similarity metric, such as cosine-similarity [138]. Doing the same operation for each requirement and test, they find traceability links and the traceability matrix.

As the indicator of success, we used precision and recall (introduced in Section 2.4.1) for each method. Obtaining high recall means finding most of the trace links, while obtaining a higher precision value in high recall ranges means finding highly relevant links correctly, hence providing better traceability in tests. The rest of this section explains how we prepared the case studies as inputs for TF-IDF, LSI and FORTA.

4.5.1 Input to FORTA

As input to FORTA, we identified requirements for each case study (Step 1 in Figure 4.4), created scenarios for each case study (Step 2 in Figure 4.4), and finally gathered execution traces for scenarios and test cases (Steps 2 and 5 in Figure 4.4 respectively) as described in Section 2.3. The relevant statistics for each case study are listed in Table 4.1.

4.5.2 Inputs to TF-IDF and LSI

To compare our approach, we implemented TF-IDF and LSI. These approaches require requirements documentations. We gathered them from their manuals and javadocs manually.

They also need the source code of the tests in text form, so we implemented a program to parse their test code using the Eclipse Java abstract syntax tree parser [7], remove the stop words (common English words like “a”, “the”, “should”), remove Java specific keywords (“public”, “static”, “final”), separate terms in Java camel-case notation (e.g. “setDocumentParser” into “document” and “parser”), and finally index the terms using Apache Lucene [4]. We then fed the results as input to these techniques as raw text. These operations were all performed programmatically, and running times are reported later in the results section.

4.5.3 Results for Requirements Traceability Links

Table 4.2 summarizes the results we obtained with each method. It contains the best precision/recall values obtained at high recall ranges. Since the first priority in traceability is retrieving all links [156], we considered recall values higher than 90%, and included in the results the best precision we could get for the recall range 90% —100%. We also included some characteristics for each methodology, such as runtime of each technique.

4.6 Discussion

Table 4.2 summarizes the case study results for FORTA, TF-IDF and LSI. We compare our results with those provided by Lormans and Van Deursen [98], where they provide precision/recall results using LSI. First, using LSI, we obtained precision/recall results similar to the ones reported by Lormans and Van Deursen [98]. Furthermore,

Table 4.2. Requirements traceability results.

Case Study	Metric	TF-IDF	LSI	FORTA
Chat System	Precision	23%	27%	99%
	Recall	99%	93%	99%
	TSP (seconds)	0.017	0.017	0.876
	TSA (seconds)	1.009	1.023	0.687
	SET (MB)	—	—	8.5
	STC (MB)	0.376	0.376	—
Apache Pool [3]	Precision	20%	20%	96%
	Recall	92%	100%	98%
	TSP (seconds)	0.023	0.023	1.110
	TSA (seconds)	1.333	1.445	0.595
	SET (MB)	—	—	12.6
	STC (MB)	1.14	1.14	—
Apache Log4j [2]	Precision	18%	22%	91%
	Recall	100%	100%	100%
	TSP (seconds)	0.022	0.022	0.770
	TSA (seconds)	1.190	1.237	0.470
	SET (MB)	—	—	10.65
	STC (MB)	0.780	0.780	—
Apache Commons CLI [1]	Precision	32%	35%	99%
	Recall	99%	95%	92%
	TSP (seconds)	0.047	0.047	1.178
	TSA (seconds)	2.045	4.510	0.221
	SET (MB)	—	—	2.22
	STC (MB)	0.728	0.728	—

TSP Time spent on preparation (in seconds)

TSA Time spent on analysis (in seconds)

SET Size of execution trace (in megabytes)

STC Size of test code (in megabytes)

Table 4.3. Scenario creation methodologies compared on Apache Commons CLI [1].

	Precision	Recall
Single scenario per requirement (as in [60, 61, 119])	99%	72%
Multiple scenarios per requirement —scenario execution units combined (on the techniques described in [60, 119])	95%	81%
Multiple scenarios per requirement —scenarios separately analyzed (FORTA)	99%	92%

our case studies confirm that LSI performs better than TF-IDF [81], since LSI can additionally match synonymous terms (different terms with the same meaning) and distinguish polysomic terms (same terms meaning different things). Furthermore, when run on the same case studies, our approach achieves higher precision on high recall levels. This is because it uses a more accurate, executable description of requirements, compared to the text similarity approach of the IR methods.

Table 4.3 provides a comparison of different ways to create scenarios for requirements. For this comparison, we chose Apache Commons CLI [1], because it had several requirements where there were multiple ways of triggering the requirement, hence a good candidate to demonstrate the importance of allowing multiple scenarios per requirement to obtain higher recall. The first row represents how existing methods operate where there is a single scenario for each requirement. This yields good precision, however recall is low because some trace links are missed. The second row represents having multiple scenarios for a requirement by combining the execution units of all scenarios into the same execution trace as if they were a single scenario (utilizing existing approaches to support multiple scenarios). In this approach, recall is higher because execution units from different scenarios are captured. However, since they are combined into a single execution trace, the execution units from different scenarios compete with each other to be chosen as feature markers, which results in slightly lower precision. Finally, the last row represents the approach FORTA supports, where multiple scenarios per requirement are allowed and each of them is used separately to choose feature markers first, and then combined altogether to represent the requirement. FORTA obtains both the same precision as the single scenario case and also the highest recall because it successfully captures different ways of triggering a requirement.

We focus on RT in tests, while the IR approaches target retrieval of RT links on a wide spectrum of artifacts such as source code, test cases and configuration files. Due

to this focus, our approach uses a technique that is suited to tests only and yields better results on test trace links compared to the IR techniques. RT in tests is an important problem and finding trace links with high precision and recall automatically has many benefits. Our approach requires some extra effort in the preparation of the scenarios initially; however, we observed that this effort was negligible because compared to test cases, the scenarios were short (2-5 lines). As an extra benefit, when created as test cases themselves, our scenarios stay up to date as software evolves, which greatly offsets the initial investment in creating them.

An advantage of our technique is that it does not require any documentation or the source code of the software. The only requirements are having executable scenarios as well as executable test cases. This proves useful for producers of commercial-off-the-shelf (COTS) software and their customers, where test cases can be provided as executables to the customers, and customers can execute their own scenarios and generate the RT matrix themselves for acceptance [44].

Another advantage of our approach is that, unlike existing research [60, 61, 119], it allows multiple scenarios for a single requirement. This helped us increase the recall rates, especially in Apache Commons CLI [1], because it had several requirements with multiple ways of being triggered.

Our approach is agnostic to the programming language used in the system, and it can be configured to work on different levels of abstraction. The only requirement is the existence of a profiler that can gather execution traces, and the execution units to be of the same type for both tests and scenarios. We implemented our tool to work for Java [68], but it can be easily extended to work for any language with a profiler available (such as C, C++, .Net, python, perl).

Our approach only applies to functional requirements currently, which is a limitation. However, existing approaches like IR can be used to complement it to detect

non-functional requirements (robustness, security, fault-tolerance), which we leave as future work.

A valuable lesson that we learned is that our technique can provide better results if, depending on the system, programming-language-specific cases are handled through some customization (e.g. polymorphism in object oriented languages). As an example, consider the class *GenericObjectPool* and the class it extends, *BaseObjectPool*, in Apache Pool [3]. We prepared scenarios using the *BaseObjectPool* class, so the execution units in the scenarios contained class and method names from *BaseObjectPool*. However, in the execution traces of the tests in *GenericObjectPool*, all traces contained *GenericObjectPool* as the class name, which caused us to miss some of the matches between scenarios' and test cases' traces. For this reason, we opted to use only the method names during the comparison of the execution units for this library and Apache Log4j [2]. This limitation can be overcome by complementing our approach with static analysis to find out class hierarchies and match the execution units more flexibly.

Another technique that can increase the accuracy of our approach is filtering out the utility execution units in the execution traces, such as utility classes or methods. In FORTA, this is partly achieved already if there are a reasonable number of features. Since we use probabilistic ranking of execution units in the feature traces, the commonly observed execution units receive a lower probability in feature marker selection. However, existing utility class/method detection techniques in the literature [73] can further complement our approach, especially when there are a small number of features. This provides more accurate feature marker selection, hence more accurate traceability links.

An important step in the scenario based methods [60, 148, 152, 151, 61, 119] is the selection of scenarios that represent the requirements. If incorrect or inaccurate scenarios are chosen, traceability results are adversely affected. Since we build upon these techniques, our method also carries the same risk. However, FORTA provides a

summary of the chosen feature markers and this risk can be minimized by investigating the feature markers and updating the chosen scenarios accordingly.

We also observed that setup and tear-down that takes place in the execution of scenarios can sometimes degrade the performance of the selection of feature markers. This is a common problem in scenario-based methods [60, 148, 152, 151, 61, 119], and solved by adding two extra features to the analysis for setup and tear-down. This way, the common execution traces for them will naturally be downgraded during feature marker selection for the other features.

Another big advantage of our approach is that, scenarios can be provided as test cases. When the source code of the system is refactored or changed, developers typically fix tests to keep them passing after the changes. Since scenarios are also test cases, developers will fix them too and scenarios will always stay up to date. While this demands some developer effort, having up-to-date requirements trace links offsets the investment. For the IR approaches, however, documentation and comments in the code may get out of date as software evolves. This results in outdated requirements traces.

4.6.1 Threats to Validity

In this section we discuss some of the issues that might have affected the results of our case studies and may limit the generalizations and interpretations of the results.

First, we cannot claim that the case studies we used represent the full extent of production software systems used in practice. Apache Pool [3], Apache Log4j [2] and Apache Commons CLI [1] are commonly used open source production software, while the Chat System is software used in a class at UCSD. We chose our case studies from different domains to mitigate this threat, which can be further reduced if more software systems of varying size from more domains are experimented with.

Second, we are not domain experts of the software used in the case studies, so

we cannot claim that we found all requirements and our scenarios are the best ones to capture them. Thus, depending on the chosen requirements and scenarios, the results may differ. Similarly, in our case studies, some tests were exercising requirements that we were not able to identify, so we did not use those tests in our analysis. Due to time constraints, we decided to use only those requirements we were able to identify, instead of going back and adding more requirements after analyzing the case study results.

Third, we prepared the ground truth for both traceability results and test intent results in our case studies manually. To mitigate risk, two different developers performed these tasks and the results were confirmed by comparing their responses. However, it is still possible that mistakes have happened.

Finally, Apache Log4j [2] had a large number of test cases, and we could not prepare the complete ground truth for it due to time constraints. So we used a randomly chosen subset of the test cases. To mitigate this factor, we chose those tests from different classes and packages in the test suite.

4.7 Conclusion

Requirements traceability is an important and active research area that aims to link requirements to different software artifacts, such as source code, test cases, and design models. Traceability has many benefits including prioritizing requirements, estimating software change impact, proving system adequacy, validation, testing and understanding the system, and finding reusable elements in the system [69, 137].

This chapter focuses on the RT problem in tests. Testing is an important step in the software development lifecycle to improve the quality of the final product. Empirical evidence shows that in many systems, the amount of testing code produced is comparable to the amount of source code of the system [149, 107]. Hence, finding traceability links in tests has gained ever more importance. Test-traceability has many benefits including

monitoring of testing accuracy and completeness with respect to requirements [135], understanding the system by looking at existing test cases as exemplars and prioritization of testing efforts [69].

In this chapter, we draw upon existing research to use features, observable units of behavior of a system that can be triggered by a user [61], to represent functional requirements of a system. Building on existing research [60, 61, 119], we trace features in source code using scenarios, executable actions that trigger features. We take these approaches further by using multiple scenarios for a single feature, treating each of them separately to find feature markers, and then combining all markers to represent a feature. This way, we do not miss trace links for features that have multiple ways of being triggered, hence increase the obtained recall values along with precision.

Once RT links in source code are gathered using scenarios, they are used to find traceability links in tests. This produces the RT links (as a requirements traceability matrix) in tests. Our approach is specifically tailored to solve the RT problem in tests, and achieves better precision/recall than the currently known approaches [18, 100, 75, 98, 106].

Our approach has many benefits: it does not require the existence of the source code or documentation of the system, it works for different programming languages and on different levels of abstraction preferred, and it is fully automated with no need for human intervention during the analysis. The only requirement is to have a profiler for the system to be analyzed.

Finally, we propose an automated process and tool support, FORTA, to create the traceability links between requirements and tests as a by-product of automated software development processes, such as TDD and continuous integration. In our approach, scenarios are also implemented as executable tests, which will be updated by developers as the code-base of the system evolves. Although this requires some effort from the

developers, unlike the IR approaches, the traceability links do not get out of date due to code and requirement changes.

Our tool, FORTA, works for Java [68] currently, however it can be easily extended to work for any language that has a profiler available.

The work in this chapter, in full, is a reprint of the material as it appears in: “Celal Ziftci and Ingolf Krueger. Tracing Requirements to Tests With High Precision and Recall. In Proceedings of the 26th International Conference on Automated Software Engineering, pages 472-475, Lawrence, Kansas, USA, November 2011. IEEE.” The dissertation author was the primary investigator and author of this paper.

4.8 Future Work

Given accurate and complete RT links, many further research possibilities exist. One is finding outdated documentation automatically, comparing the RTMs produced by FORTA and the IR approaches. Another area is to use the RT links to provide prioritization of the analysis of failing tests for a developer/tester.

Chapter 5

Test Intents: Enhancing the Semantics of Requirements Traceability Links in Test Cases

In this chapter, we focus on requirements traceability in tests, and build on existing techniques on finding requirements traces in test cases (introduced in Chapter 4) to find **test intents**, i.e. which requirements test cases aim to test. Test intents further enhance the requirements traceability links by showing which links are the intended targets of a test case and provide richer information for stakeholders.

5.1 Introduction

As discussed in Section 2.2.5, requirements traces in tests demonstrate to the customers which requirements are tested to achieve reasonable quality [135]. For product developers, testers and managers, they show which requirements are covered by which test cases. For new developers joining a team, or customers of the software, they provide an overview of which test cases to examine for comprehension of the system. For a testing lead, they help prioritize testing efforts [69].

These testing and software quality related activities are possible only if requirements traces are accurate, complete and up to date. As discussed in the previous chapter

(Chapter 4), there are existing techniques for requirements tracing in test cases that address the problem of finding such accurate and complete traces [18, 100, 74, 98, 106].

Our work builds upon these existing techniques on finding requirements traces in tests, and enhances the found trace links by highlighting the intended requirements a test case was implemented to test. This makes the tasks mentioned above easier to perform for stakeholders and increases the benefits of having traceability links for test cases.

5.2 Background

As discussed in Section 2.2.15, requirements traces are typically visualized using a traceability matrix where rows and columns represent two different sets of artifacts. The RTM for our Chat System is given in Table 2.3.

This table is useful, in that, it shows which test cases exercise which requirements, hence are related to which requirements. However, given this information, it is not very obvious what a test is really aiming to test [94].

Consider the following example for the Chat System in Table 5.1: the test `testSendMessage` intends to test the `send-message` requirement (as described in its name). However, `connect` and `sign-on` to the server are required before `send-message` can be performed. For this reason, `testSendMessage` exercises them, too. Hence, the RTM will contain links between this test and all three requirements. This is certainly valuable information; however it would be more useful for humans if the real aim of the test, called **test intent** (`send-message` in this case), is revealed.

We propose that developers typically implement test cases to test that a requirement (or the interaction of multiple requirements) is satisfied. Hence, in a test case, they would typically not exercise requirements that are not necessarily contributing to the requirement to be tested since that would make the test case less readable, it would cause parts of the test case to be redundant and it would actually mean doing more

Table 5.1. Sample Requirements Traceability Matrix for Tests of the Chat System (duplicate of Table 2.3 for convenience).

	connect	sign-on	send-message	sign-off
t_1 : testConnect	✓			
t_2 : testConnectAndSignOn	✓	✓		
t_3 : testSendMessage	✓	✓	✓	
t_4 : testSignOff	✓	✓		✓

Table 5.2. Sample Requirements Traceability Matrix for tests enhanced to show test intents for the Chat System.

	connect	sign-on	send-message	sign-off
t_1 : testConnect	✓✓			
t_2 : testConnectAndSignOn	✓	✓✓		
t_3 : testSendMessage	✓	✓	✓✓	
t_4 : testSignOff	✓	✓		✓✓

work than required to test that specific requirement. In the example above, a developer would not necessarily exercise connect in testSendMessage unless it was required or it contributed to the test case in some way. This observation is the main motivation behind finding test intents. We also propose to take the next step and think that the RTM (or a filtered version of it) should include only the test intents, because conceptually that is what developers are interested in in the RTM. In the example above, since testSendMessage aims to test send-message, it would be beneficial for a developer to see that as the most important traceability link in the RTM.

As examples, Tables 5.2 and 5.3 show enhanced versions of the RTM for test

Table 5.3. Sample Requirements Traceability Matrix for tests enhanced to rank requirements according to likelihood to be test Intents for the Chat System.

	connect	sign-on	send-message	sign-off
t_1 : testConnect	(1)			
t_2 : testConnectAndSignOn	(2)	(1)		
t_3 : testSendMessage	(3)	(2)	(1)	
t_4 : testSignOff	(3)	(2)		(1)

cases. In Table 5.2, the RTM is provided such that the proposed intents are marked with double ticks and in blue color. In Table 5.3, even more information is provided, where the requirements are ranked according to their likelihood of being the intent of the test case: (1) means that the requirement is the highest likely intent of the test case, while larger numbers mean the test is less likely to be directly targeting to test a requirement. The representation shown in Table 5.3 can be especially useful when test cases test the interaction of multiple requirements. These enhanced representations of the RTM provide richer information to stakeholders that might be useful for maintenance tasks described in Section 5.1.

5.3 Contribution

In this chapter, we use the traceability links between requirements and test cases as input, and find test intents. Our work makes the following contributions:

- A method for identifying which requirement a test case is specifically targeting to cover/test (i.e. its intent).
- An automated tool to find test intents, given trace links between requirements and test cases.

5.4 Finding Test Intents

Figure 5.1 shows the input, process and output of our technique. Our technique assumes the existence of trace links between requirements and test cases. To obtain these, one of the existing techniques in the literature can be used [18, 100, 75, 98, 106]. Although the trace links can be provided in any format, for the sake of simplicity, we assume an RTM is given since it is a very common representation for trace links (the input in Figure 5.1).



Figure 5.1. The steps in finding test intents.

We propose finding test intents by applying formal concept analysis (FCA) [28, 66] to the RTM (the middle step in Figure 5.1). We propose using FCA because it can provide a ranking of exercised requirements for each test case from more specific to less specific (i.e. it can rank attributes of a binary relation with respect to objects). We propose that requirements more specific to a test case are more likely to be the intent of a test case than the less specific requirements. With this reasoning, in our Chat System example in Table 5.1, we expect `send-message` to be ranked as more specific for `testSendMessage` than `connect` and `sign-on`, because it is likely that there are other test cases implemented to test `connect` and `sign-on`, hence these requirements will be considered less specific for `testSendMessage`. Therefore, `send-message` would be more likely to be the intent for `testSendMessage` (as shown in Table 5.3).

FCA may not work very well if the system has many test cases that test the same requirement, because it would be hard to rank the requirements from more to less specific since there would not be enough data to make the distinction between those similar test cases in the concept lattice. However, if this is not the case, then FCA would be expected to perform well. The following subsections provide an overview of FCA and how we use it to find test intents.

5.4.1 Formal Concept Analysis

FCA is an automated and principled way of deriving an ontology from a set of objects and their attributes (properties). The foundations of FCA were laid by Birkhoff [28] and the term “formal concept analysis” was coined by Wille [66], who explored its mathematical foundations.

FCA works on a relation $I \subseteq O \times A$ between a set of objects O and a set of attributes A . The tuple $C = (O, A, I)$ is called a **formal context**. Table 5.1 shows a sample formal context for objects:

`{testConnect, testConnectAndSignOn, testSendMessage, testSignOff}`

and attributes:

`{connect, sign-on, send-message, sign-off}`

For a set of objects $O \subseteq O'$, the set of common attributes $attr(O')$ is defined as:

$$attr(O') = \{a \in A \mid (o, a) \in I, \forall o \in O'\} \quad (5.1)$$

For a set of attributes $A' \subseteq A$, the set of common objects $obj(A')$ is defined as:

$$obj(A') = \{o \in O \mid (o, a) \in I, \forall a \in A'\} \quad (5.2)$$

A tuple $c = (O_1, A_1)$ is called a **concept** if and only if $A_1 = attr(O_1)$ and $O_1 = obj(A_1)$, i.e. all objects in O_1 share all attributes in A_1 and no other objects with all attributes in A_1 exist. Table 5.4 shows all concepts that exist in the formal context shown in Table 5.1. As an example, $c_2 = (\{t_2, t_3, t_4\}, \{\text{connect, sign-on}\})$ is a concept since the tests t_2, t_3 ,

Table 5.4. Concepts in the formal context for the Chat System RTM given in Table 5.1

Concept	(objects, attributes)
c_1	$(\{t_1, t_2, t_3, t_4\}, \{\text{connect}\})$
c_2	$(\{t_2, t_3, t_4\}, \{\text{connect}, \text{sign-on}\})$
c_3	$(\{t_4\}, \{\text{connect}, \text{sign-on}, \text{sign-off}\})$
c_4	$(\{t_3\}, \{\text{connect}, \text{sign-on}, \text{send-message}\})$
c_5	$(\emptyset, \{\text{connect}, \text{sign-on}, \text{send-message}, \text{sign-off}\})$

and t_4 all exercise `connect` and `sign-on`, and they are the only tests that exercise both `connect` and `sign-on`.

The set of all concepts of a given formal context forms a partial order with the ordering operator \leq :

$$c_1 \leq c_2 \leftrightarrow O_1 \subseteq O_2 \ \& \ A_2 \subseteq A_1 \quad (5.3)$$

where:

$$c_1 = (O_1, A_1)$$

$$c_2 = (O_2, A_2)$$

For $c_1 \leq c_2$, we call c_1 a **sub-concept** of c_2 , and c_2 a **super-concept** of c_1 .

The set of all concepts L for a formal context and the partial ordering \leq form a complete lattice, called a **concept lattice**. The most general concept is called the **top element**, and the most special concept the **bottom element**. Figure 5.2 shows the concept lattice for the formal context in Table 5.4. The top element in the lattice is:

$$c_1 = (\{t_1, t_2, t_3, t_4\}, \{\text{connect}\})$$

since it is the most general one (i.e. it contains all of the objects). The bottom element in the lattice is:

$$c_5 = (\emptyset, \{\text{connect}, \text{sign-on}, \text{send-message}, \text{sign-off}\})$$

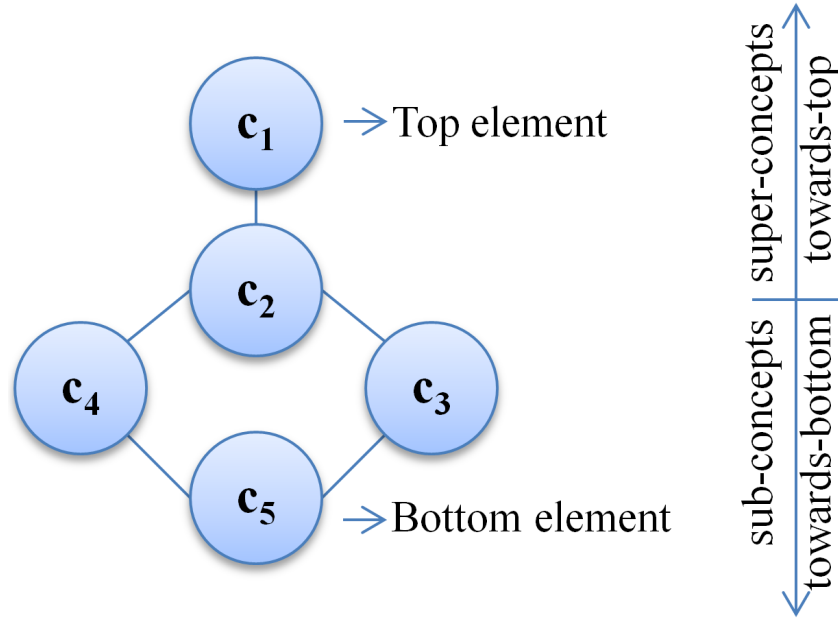


Figure 5.2. Concept lattice for the formal context shown in Table 5.4.

since it is the most special concept (i.e. it requires all attributes).

For a more in depth analysis of the mathematical foundations of FCA, we refer the reader to [28].

5.4.2 Finding Test Intents Using FCA

To find test intents, as demonstrated in the previous subsection, we use the test cases as objects, and requirements as attributes of a formal context in FCA (see Table 5.4 and Figure 5.2). For each test case, we find the most specific concept(s) in the concept lattice, and walk up until the top element. The attributes observed during this traversal operation provide us the ranking of requirements from most specific to least specific for a given test.

As an example, consider the test t_3 : `testSendMessage` in Table 5.1. The most specific concept for t_3 in the lattice (in Figure 5.2) is:

$$c_4 = (\{t_3\}, \{\text{connect, sign-on, send-message}\}).$$

Walking up the lattice towards the top element, starting from c_4 ,

$$c_2 = (\{t_2, t_3, t_4\}, \{\text{connect}, \text{sign-on}\})$$

and

$$c_1 = (\{t_1, t_2, t_3, t_4\}, \{\text{connect}\})$$

are its super-concepts in traversal order. During each step of the traversal, we find the attributes that exist in the sub-concept and do not exist in the super-concept. We take these attributes as more specific for the given test case. As an example, `send-message` is the only attribute that exists in c_4 but not in c_2 . The attribute `send-message` is therefore most specific for t_3 , and `sign-on` and `connect` are less specific in the given order. Hence the requirements are ranked, from higher to lower, as `send-message`, `sign-on`, and `connect`, as shown in Table 5.3 for `testSendMessage`.

Note that, it is possible that there are multiple concepts that can be considered as most specific for a test case. In such a case, we traverse the concepts starting from the lowest level towards the top element. If two concepts are on the same level in the lattice, we assign no preference between them in terms of ranking (we assume they rank the same).

Furthermore, if two attributes appear to be most specific for a test at the same time, i.e. if there are at least two attributes that are different between a concept and its parent during traversal, we again assume those attributes rank the same for the given test case.

5.4.3 Tool Support

We provide tool support for our approach, which can be integrated into automated software processes such as continuous build and TDD. Given the RTM, our tool outputs the enhanced RTMs that show the real intent of each test, as shown in Tables 5.2 and 5.3.

Table 5.5. Case study properties to evaluate finding test intents.

Case Study	# Requirements	# Test Cases
Chat System	16	20
Apache Pool [3]	16	77
Apache Log4j [2]	10	69
Apache Commons CLI [1]	11	181

5.5 Evaluation

To assess the validity of our approach, we used all of our case studies described in Section 2.3. We found the requirements for each case study as described in Section 2.3. Table 5.5 lists the statistics of the case studies relevant to our evaluation. The rest of this section explains how we prepared the input for our technique.

5.5.1 Finding the Baseline for Test Intents

We prepared a baseline for each case study to assess the quality of our technique. We manually investigated each test case in the test suite, and recorded the test intent for each.

We asked two different developers to get themselves familiar with each case study and find the test intents of each test case independently. We then compared their findings to form the baselines. Whenever they had disagreement on the intent of a test case, we excluded that test from our baseline. Whenever both developers had the same findings for a test case, we included that test case in our baseline. Table 5.6 presents how many test cases we used in the baseline and summarizes the results of the intent mining process for each case study.

In our tool, we used a parameter T that specifies how many of the suggested requirement names a human would be willing to look at to find out about the real intent of a test. As an example from our Chat System, consider a test t_1 with a clear intention

Table 5.6. Test intent mining results.

Case Study	# Tests in Baseline	% Intents Correctly Identified		
		$T = 1$	$T = 2$	$T = 3$
Chat System	15	93%	100%	100%
Apache Pool [3]	44	33%	60%	79%
Apache Log4j [2]	34	62%	91%	100%
Apache Commons CLI [1]	117	50%	81%	97%

of testing the `sign-off` requirement, while also exercising `connect` and `sign-on`. We observed that our ranking tool suggests the following ranking sorted from higher to lower priority: $\{\text{sign-on}, \text{sign-off}, \text{connect}\}$. In this case, if $T = 1$, this corresponds to a human only willing to look at the top suggestions. Hence, our tool will have missed the intent because it provides `sign-on` as the highest ranked requirement, whereas `sign-off` is the real intent of the test. However, if the human were willing to look at the top two suggestions, i.e. $T = 2$, then our tool would be considered to have made a correct suggestion since `sign-off` is among the top two suggestions returned by our tool.

5.6 Discussion

Table 5.6 summarizes the test intent results our technique produces. Given these results, we can correctly identify the real intent of a test case with a high accuracy within two to three suggestions for each test case. This greatly reduces the amount of analysis a human would need to do on the source code of a test to understand what it aims to test. We discuss advantages and limitations of our approach in the rest of this section.

An advantage of our technique is that it does not require any documentation or the source code of the software. The only requirement is to have the trace links between requirements and test cases. These links can be obtained by existing techniques in the literature [18, 100, 75, 98, 106]. Since we build upon these existing techniques, the success of our method depends on the success of these methods. For our method, it may

be beneficial to combine the use of multiple techniques that find requirements traces so that likelihood of errors in the found trace links is lower, and our technique yields better results.

On mining the intent of test cases, using FCA may not work well if many test cases are implemented to test very similar requirements. In such a scenario, FCA may not be able to find specific requirements for each test. The lower success rate in Apache Pool [3] can be attributed to this fact. However if this is not the case (as in the other case studies), then FCA is expected to perform well, because test cases are typically linked to specific requirements, and they explicitly target that requirement, and do not exercise the other requirements unless they have to (unless there is a dependency of a requirement to another one).

Our approach is agnostic to the programming language used in the system. As long as the traceability links are provided, it can work for any system.

Our approach applies to both functional and non-functional requirements, as long as they are contained in the traceability links provided in the input RTM.

5.6.1 Threats to Validity

In this section we discuss some of the issues that might have affected the results of our case studies and may limit the generalizations and interpretations of the results.

First, we cannot claim that the case studies we used represent the full extent of production software systems used in practice. Apache Pool [3], Apache Log4j [2] and Apache Commons CLI [1] are commonly used open source production software, while the Chat System is software used in a class at UCSD. We chose our case studies from different domains to mitigate this threat, which can be further reduced if more software systems of varying size from more domains are experimented with.

Second, we are not domain experts of the software used in the case studies,

so we cannot claim that we found all requirements. Thus, depending on the chosen requirements, the results may differ.

Third, we prepared the baselines in our case studies manually. To mitigate risk, two different developers performed these tasks and the results were confirmed by comparing their responses. However, it is still possible that mistakes have happened.

5.7 Related Work

Our work builds upon existing techniques on requirements traceability in test cases, since it assumes the existence of the RTM for test cases. Related work on this topic has already been discussed in Section 4.2.

5.8 Conclusion

Requirements traceability is an important and active research area that aims to link requirements to different software artifacts, such as source code, test cases, and design models. Traceability has many benefits including prioritizing requirements, estimating software change impact, proving system adequacy, validation, testing and understanding the system, and finding reusable elements in the system [69, 137].

This chapter focuses on the RT problem in tests. Testing is an important step in the software development lifecycle to improve the quality of the final product. Empirical evidence shows that in many systems, the amount of testing code produced is comparable to the amount of source code of the system [149, 107]. Hence, finding traceability links in tests has gained ever more importance. Test-traceability has many benefits including monitoring of testing accuracy and completeness with respect to requirements [135], understanding the system by looking at existing test cases as exemplars and prioritization of testing efforts [69].

In this chapter, we propose that providing an enhanced RTM is beneficial for stakeholders and propose a novel method for identifying which requirement a test case is specifically targeting to cover/test. Our approach mines the intent of such tests from a given set of requirements trace links on test cases by ranking the requirements a test case is exercising and providing humans with an easily identifiable ranked suggestion list of requirements for each test case. We propose to take the next step and think that the RTM (or a variant of the RTM) should include only the test intents, because conceptually that is what developers are interested in in the RTM.

Finally, we provide tool support to find test intents as a by-product of automated software development processes, such as TDD and continuous integration.

The work in this chapter, in full, is a reprint of the material as it appears in: “Celal Ziftci and Ingolf Krüger. Test Intents: Enhancing the Semantics of Requirements Traceability Links in Test Cases. In Proceedings of the 28th ACM Symposium On Applied Computing, pages 1272-1277, Coimbra, Portugal, 2013. ACM.” The dissertation author was the primary investigator and author of this paper.

5.9 Future Work

In test cases that test the interaction of multiple requirements, if such interactions can first be detected, our technique can be improved to provide better rankings.

Given accurate and complete test intents, many further research possibilities exist: using the test intents to provide prioritization of the analysis of failing tests; and minimization of test suites based on test intents.

Chapter 6

Automatically Mining Requirements Relationships From Test Cases

Requirements relationships express conceptual dependencies, constraints and associations among the requirements of a software system [122]. Examples of these relationships are dependencies, a requirement needed by another one; and hint-relations, a requirement being frequently used in conjunction with another requirement [122, 47, 36] (hence, when used, it hints at the other requirement).

In this chapter, we propose a new technique to automatically mine requirements relationships from existing test cases using features. This technique builds on the work in Chapter 4, and it works under the assumed relationships between requirements and features described in Section 2.2.4.

6.1 Introduction

Requirements relationships represent different types of associations between requirements [122]. Two common types of such associations are dependency and hint-relation [122, 47].

Definition 6.1.1 (Requirement Dependency). A requirement dependency is a tuple (r_1, r_2) such that r_2 depends on r_1 if and only if r_1 must be exercised before r_2 for r_2 to

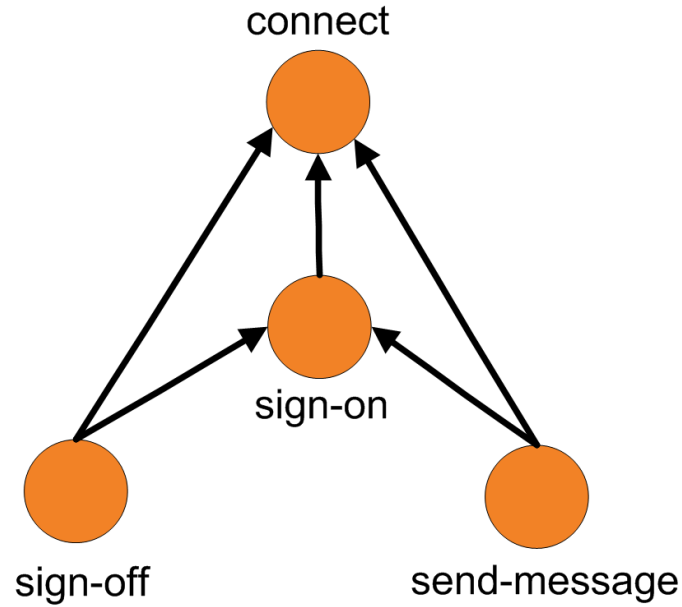


Figure 6.1. Dependencies between requirements of the Chat System.

behave correctly.

An example from our Chat System is the dependency relationship between requirements `connect`, `sign-on`, `send-message` and `sign-off` (shown in Figure 6.1): `send-message` requires `sign-on`, and `sign-on` requires `connect`, because to sign on and create a session, one needs to first connect to the server; and to send a message to another chat client, one needs to first sign on successfully. Similarly, `sign-off` depends on `sign-on` and `connect`.

Definition 6.1.2 (Requirement Hint-Relation). A requirement hint-relation is a tuple (r_1, r_2) such that r_1 and r_2 are in a hint-relation if and only if (r_1, r_2) is not a requirement dependency, and r_1 and r_2 are typically used together.

Using hint-relations, understanding a requirement potentially helps understanding the other, and it becomes easier to estimate the impact of change in one requirement on the other requirements.

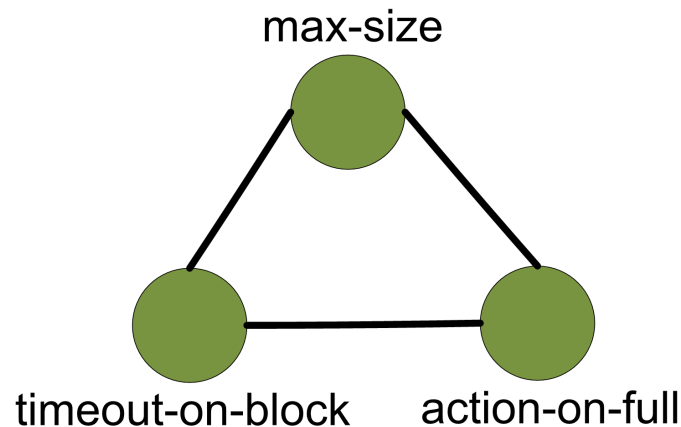


Figure 6.2. Hint-relations between some of the requirements of Apache Pool [3].

As an example, consider Apache Pool [3] (one of the case studies we used). This is a library that provides resource pooling, where a limited number of resources exist and the customers of the pool can check-out and check-in objects as they need them. It is typically used to pool resources that are expensive to create and destroy (such as database connections), to avoid the overhead associated with those lifecycle activities. Figure 6.2 shows hint-relations between some of the requirements of this library. Pool provides a maximum size to limit the number of resources (`max-size` requirement). One can decide on what happens when the pool is full and a consumer asks to check-out a resource, the `action-on-full` requirement, (such as block the consumer call, or throw an exception). Similarly, if “block the consumer’s call” is the action chosen, then the pool provides a way to specify for how long to block the call (`timeout-on-block` requirement). These requirements are not strictly dependent on each other, i.e. setting one does not necessarily require setting the others (since there are defaults for each of them). However, they are closely related to each other, and when a developer tries to understand one of them, knowing that these requirements are related can help him during comprehension, since it is expected that the implementation of these requirements are related to and possibly interleaved with each other.

Identification of requirements relationships in a system's domain is important in several aspects. During the design and maintenance of a system, understanding requirements in isolation is typically not enough, since requirements interact with or depend on each other, and making a change on one requirement may have impact on other requirements [27, 33, 91, 36, 124].

During the design phase, requirements relationships help ensure that the system will behave correctly by exposing potential conflicts between requirements to designers [60]. During maintenance, they help developers determine if a change on the system will create conflicts that didn't exist before [60, 33, 91]. In all stages of the software lifecycle, they help developers comprehend a requirement along with its dependencies and the other requirements it typically interacts with before a maintenance task is performed [153, 80, 117, 124]. Finally, during all stages of the software lifecycle, they help customers see an overview of requirements, support design and architecture decisions about the evolution of the software system and prioritize development efforts [143, 136, 131, 36].

Requirements relationships are typically modeled during the requirements analysis stage. Similar to other software artifacts, such as source code, test cases and documentation, the system's requirements relationships model evolves during maintenance of and updates to the system. The benefits of the knowledge about requirements relationships, described above, are possible only if the requirements relationships model is accurate and up-to-date. Requirements relationship models can be manually maintained as software evolves. However, it is labor-intensive, time-consuming and error-prone to acquire and maintain such software artifacts. Furthermore, without disciplined developers, the model gets out-of-date over time during maintenance [69, 99, 34]. Therefore, retrieving and maintaining the relationships model via an automated process is important and convenient.

In this chapter, we propose to automatically reverse engineer requirements re-

relationships using existing test cases, such as unit tests, integration tests, acceptance tests and system tests. Since the effort spent on testing and its cost in the development process is substantial [107, 149, 74], we propose using this investment to reverse engineer requirements relationships from the test cases automatically.

6.2 Related Work

Previous research exists on automatically finding requirements relationships. There are approaches that use scenarios to exercise requirements on the actual system [130, 95]. These approaches execute the scenarios of the requirements on the system. While scenarios are executed, they track object instantiations [130] and object aliasing [95], i.e. how objects are passed in the system between components (classes, methods) using a profiler (or a similar technology). These approaches propose that if an object that is created during the execution of a requirement is used in another requirement during its execution, then there is a direct dependency of the latter requirement on the former. These methods are successful in finding requirements dependencies. However, they have some shortcomings. First, since they track objects, they only work for systems implemented in object oriented languages; not for other types of languages such as functional or procedural languages. Second, they are sensitive to the implementation of the system since they rely on the flow of objects. They may report non-existing dependencies and hint-relations due to sharing of non-critical utility objects in the system, and they may miss some relationships since not all relationships require sharing or flow of objects in the system.

A different family of approaches to automatically retrieve requirements relationships also uses scenarios to represent requirements. Similar to the previous ones, they execute the scenarios and collect execution traces as they execute [60, 117]. Unlike the previous methods, they trace the components executed, such as class and method

names. They propose that if the components observed in the execution of a requirement are a subset of the components observed in another requirement, the latter requirement is said to have a dependency on the former one. An advantage of these approaches is that they not only work on object oriented systems, but also other types of systems, because they don't use object instantiation and flow to determine dependency. One of the disadvantages of these approaches is that observing the same components may not be sufficient to conclude the existence of a dependency. If the common methods for the first requirement are executed in a different order than the second one, this may point to a different requirement. Therefore, these approaches can be misguided, since they only analyze the existence of the components, not their order. Another disadvantage of these approaches is that they only detect dependencies; they cannot detect hint-relations between requirements.

A common disadvantage to both of these approaches is in the way they use scenarios. They require executable scenarios for each requirement, which, in some systems, do not readily exist. Furthermore, unless they are supported with manual effort (as in [60]), these approaches can retrieve trace links only for functional requirements of the system. Finally, these approaches might miss some of the trace links, because they only gather trace links on those parts of the system that are exercised by the scenarios. These approaches assume representing each requirement with a single scenario. However, some requirements might be triggered in multiple ways. As an example, consider the Chat System: users are provided a graphical user interface, a command-line client and programmatic access to the server. These approaches will need to choose only one of these as a scenario. Therefore, it is inevitable to miss some trace links for some requirements. This shortcoming can be gapped using our approach as described in Section 4.4. In this approach, a single requirement can be represented with the use of multiple scenarios, which avoids missing some requirements trace links.

Another disadvantage of these approaches is that they are very sensitive to the selection of scenarios. As an example, consider two of the Chat System's requirements: `connect` and `sign-on`. As explained earlier, `sign-on` depends on `connect`. During the selection of scenarios, the scenario for `sign-on` should encapsulate the actions for `connect`, too. Otherwise, neither the objects nor the components executed during its execution will contain the objects or components for `connect`. Therefore, the described approaches will fail to detect the dependency. We propose that this makes these approaches very sensitive to the selection of scenarios.

The method that we propose in this chapter builds on existing dynamic analysis based requirements tracing methods to automatically find requirements relationships. We already discussed the relevant body of research on requirements tracing and dynamic analysis in Section 4.2.

6.3 Contribution

In this chapter, similar to the recent methods described in Section 6.1, we use scenarios to trigger features, extract feature markers (execution units that can represent each feature), observe the feature markers in test cases as they execute, and find requirements relationships automatically. Our work makes the following contributions:

- A new method to find dependencies and hint-relations between requirements. Our method performs as good as or better than existing techniques on our case studies.
- A new method to find dependencies and hint-relations between requirements that is broader in applicability. Unlike existing methods, our method is not sensitive to the selection of scenarios used to exercise requirements. Furthermore, unlike existing methods, our method is not limited to object oriented systems; it works for all systems with a profiler available.

- An automated process and tool support to reverse engineer requirements relationships from a software system's test cases as a by-product of automated software development processes, such as TDD and continuous integration.

6.4 REQRELEX: Mining Requirements Relationships from Test Cases

The work in this chapter partially builds upon previous work on tracing requirements in test cases described in Section 4.4.

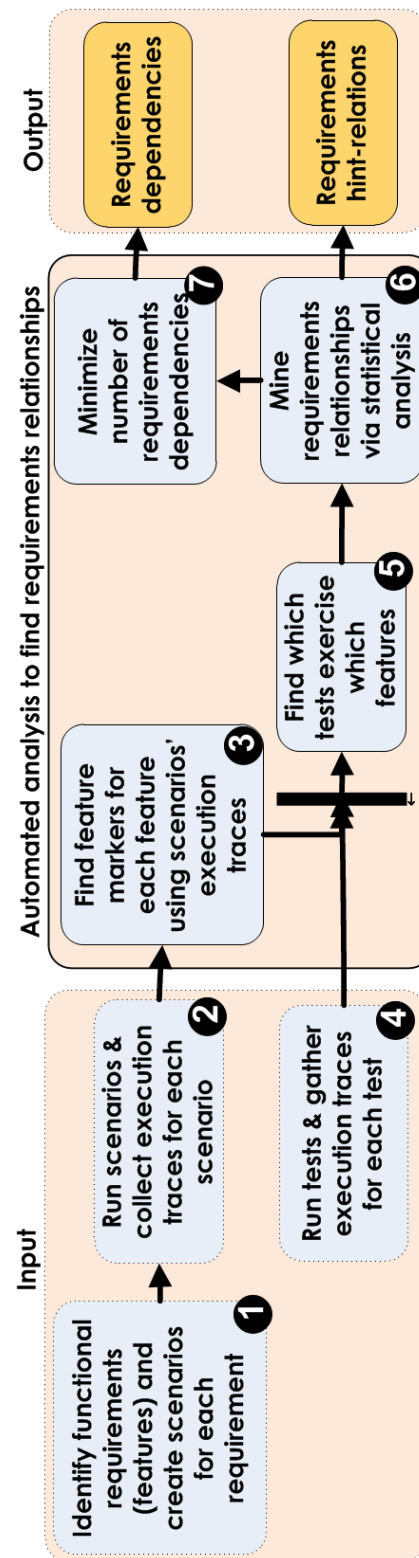


Figure 6.3. Inputs, outputs and steps of REQRELEX. Execution traces of the scenarios and test cases are the inputs, while the requirements dependencies and hint-relations are the outputs.

Figure 6.3 summarizes the inputs, flow and outputs of our approach. We implemented a tool, REQRELEX —Requirements Relationship Extractor, to automate this process. The first five steps in Figure 6.3 are identical to the steps detailed in Figure 4.4. Steps after those are specific to the work described in this chapter. The rest of this section explains those steps specific to the work in this chapter in more detail.

6.4.1 Mining Requirements Relationships

Once we find out which test cases execute which features (as described in Sections 4.4.1, 4.4.2, 4.4.3), we investigate the order in which features are observed in each test case (Step 6 in Figure 6.3).

Based on this, we propose the following:

- If a feature f_p is often observed before another feature f_q , we deduce that it is highly likely that f_q depends on f_p .
- If two features f_p and f_q are often executed in the same test case (but in possibly different orders in different test cases), we deduce that it is highly likely that f_p and f_q have a hint-relation, i.e. they do not depend on each other, but they are typically used together.

As an example, consider the feature markers for `connect` and `sign-on`, and the execution trace for the test `testConnectAndSignOn`:

`connect` : $[m_2]$

`sign-on` : $[m_4]$

`testConnectAndSignOn` : $(m_2, m_4, m_8, m_{16}, m_{46})$

We observe that in `testConnectAndSignOn`, `connect` is executed before `sign-on`

(because m_2 comes before m_4). If this is observed in many other test cases, we propose that sign-on likely depends on connect. Similarly, if two features are observed in many test cases, but in different orders (i.e. in some, one of them precedes the other, while in others the reverse happens), we propose that they likely have a hint-relation. In summary, to find such relationships between requirements, we analyze the execution traces of all test cases and look for statistically significant correlations between features observed together in test cases (Step 6 in Figure 6.3).

For readers with a background in data mining, the technique we use here is a modified version of the *Apriori Method*[14] used to perform *frequent item set mining*[13] where we only count item sets of size two and we also take into consideration the order of items to be mined.

6.4.2 Minimizing the Number of Requirements Dependencies

Once this analysis is performed on the execution traces of test cases, there will likely be many requirements dependencies discovered due to the transitive nature of dependence. As an example, consider the requirements from our Chat System: connect, sign-on, send-message. As described earlier, send-message depends on sign-on, which depends on connect. In such a dependence relationship, it is not necessary to explicitly document that send-message depends on connect, since that is already implied due to transitivity. In our analysis of the test cases, there will be many such dependencies discovered explicitly (due to the nature of the analysis performed in Section 6.4.1). Unless such implicit transitive relationships are discarded, there will be an overwhelming amount of information for stakeholders to consume. For this reason, we build a graph on the requirements dependencies we discover, and we apply transitive graph reduction to discard the implicit dependencies (Step 7 in Figure 6.3). This provides a much clearer picture of the dependencies for developers (see Figure 6.4 for an example

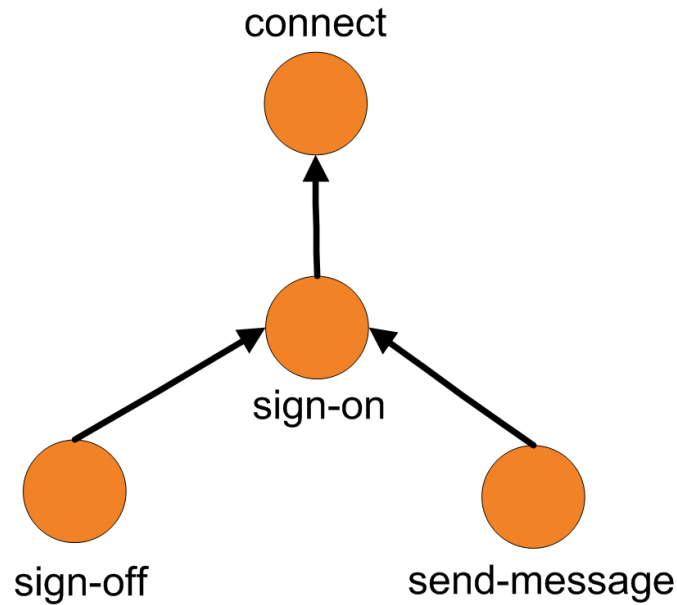


Figure 6.4. Sample results on minimizing the number of dependencies found in the Chat System via transitive reduction. This graph is a reduced version of the one shown in Figure 6.1.

of how the dependencies in Figure 6.1 look like after reduction).

This reduction is not performed for hint-relations, since hint-relation is not transitive like dependence.

6.4.3 Tool Support

For our approach, we provide automated tool support that can be integrated into automated tool can be run as a task similar to running test cases and providing code coverage in a continuous build system. Execution traces for each feature and execution traces for tests are the inputs to our system. Given these inputs, our tool outputs the requirements relationships. For an example, see Figure 6.4 where a portion of the automatically retrieved requirements dependencies are output by REQRELEX for the Chat System.

Table 6.1. REQRELEX case study properties.

Case Study	# Total Relationships	# Dependencies	# Hint-Relations
Chat System	28	26	2
Apache Pool [3]	9	3	6
Apache Commons CLI [1]	21	3	18

6.5 Evaluation

To assess the validity of our approach, we used three of our case studies discussed in Section 2.3: the UCSD Chat System, Apache Pool [3], and Apache Commons CLI [1]. Table 6.1 summarizes the statistics relevant to requirements relationships for each project. To find the baseline for our evaluation, we asked two developers to find the dependencies and hint-relations for each case study, and compared their responses. When there was a conflict, we left out that relationship from our baseline.

To compare our results, we implemented two recent techniques for automatic detection of requirements relationships: object flow analysis of Salah and Mancoridis [130] and Lienhard et al. [95] (which we call Salah’s approach in the rest of this chapter), and scenario analysis of Egyed [60] (which we call Egyed’s approach in the rest of this chapter). Salah’s approach [130, 95] tracks the flow of objects as scenarios are executed and proposes that: if a feature uses objects created by another feature, there is a dependency between them; if two features share usage of some objects, they have a hint-relation. Egyed’s approach [60] analyzes the scenarios themselves: if the scenario of a feature contains all components of another feature, then the former depends on the latter.

The next two sections explain the input preparations of the case studies for REQRELEX, Salah’s approach [130, 95] and Egyed’s approach [60].

6.5.1 Input to REQRELEX

As input to REQRELEX, we identified requirements for each case study (Step 1 in Figure 6.3), created scenarios for each case study (Step 1 in Figure 6.3), and finally gathered execution traces for scenarios and test cases (Steps 2 and 4 in Figure 4.4 respectively) as described in Section 2.3.

6.5.2 Inputs to Other Approaches

For Salah’s approach [130, 95], we tracked objects when they are instantiated and then used elsewhere using their location in the heap. This ensures that we follow both the flow of objects [130] and any aliasing effects [95].

For Egyed’s approach [60], we used the same scenarios and their execution traces that we prepared as input to REQRELEX.

6.5.3 Evaluation Criteria

We use precision, recall and the f-measure introduced in Section 2.4.1 as the metrics to measure the success of each technique. In the context of finding requirements relationships, the “relevant” relationships are the actual ones, i.e. the ones that we try to find. “Retrieved” relationships are what an approach suggests. Precision and recall correspond, respectively, to accuracy and completeness of the retrieved requirements relationships compared to the relevant relationships. On finding requirements relationships, it is important to obtain a high f-measure, because that implies finding both a low number of false positives and a low number of false negatives.

6.6 Discussion

Table 6.2 summarizes the case study results for all approaches: Salah’s approach [130, 95], Egyed’s approach [60] and REQRELEX. We provide precision, recall and

f-measure values for each approach on the respective types of requirements relationships they support. For REQRELEX and Salah's approach, we provide results for both requirements dependencies and hint-relations as well as a combination of the two (combined). For Egyed's approach, we only provide requirements dependency results, since it only provides dependency relationships for requirements. During our comparisons, we use the f-measure since it incorporates both precision and recall with equal weight.

Table 6.2. Case study results on finding requirements relationships

Case Study	Technique	Combined			RESULTS				Dependency		
		Prec- ision	Rec- all	F- measure	Prec- ision	Rec- all	F- measure	Prec- ision	Rec- all	Prec- ision	F- measure
Chat System	1. Salah	40.43	67.86	50.67	5.56	100	10.53	100	57.69	73.17	
	2. Egyed	—	—	—	—	—	—	100	100	100	
	3. REQRELEX	78.79	92.86	85.25	0	0	0	100	96.15	98.04	
	4. Egyed*	—	—	—	—	—	—	0	0	0	
	5. REQRELEX *	73.68	100	84.85	0	0	0	89.66	100	94.55	
Apache Pool [3]	1. Salah	6.1	62.5	11.12	3.66	50	6.82	33.33	100	50	
	2. Egyed	—	—	—	—	—	—	66.67	100	80	
	3. REQRELEX	30.43	87.5	45.16	25	83.33	38.46	66.67	100	80	
	4. Egyed*	—	—	—	—	—	—	0	0	0	
	5. REQRELEX *	30.43	87.5	45.16	25	83.33	38.46	66.67	100	80	
Apache Commons CLI [1]	1. Salah	66.67	19.05	29.63	66.67	22.22	33.33	0	0	0	
	2. Egyed	—	—	—	—	—	—	23.08	100	37.5	
	3. REQRELEX	37.14	61.9	46.43	28	38.89	32.56	30	100	46.15	
	4. Egyed*	—	—	—	—	—	—	0	0	0	
	5. REQRELEX *	37.14	61.9	46.43	28	38.89	32.56	30	100	46.15	

Table 6.2 contains the results of two different experiments we performed: In the first experiment (the first three rows for each case study), the inputs are prepared in conformance to what each technique expects as described in Section 6.4. In the second experiment (rows four and five for each case study, where each technique is marked with a *), the input is modified to demonstrate a shortcoming of the existing techniques on scenario selection. Each experiment is discussed in detail below.

As discussed in the related work section, Salah’s approach [130, 95] might produce false positives since object sharing or flow does not always imply that there are relationships between requirements (but rather in their implementations). It may also result in false negatives, since requirements relationships may exist in systems even in the absence of object interactions. These are observed in our case studies (since precision and recall are not equal to 100%). Egyed’s approach [60] should not have false positives if scenarios are properly selected (hence 100% recall). However, it can produce false positives, since it only investigates the existence of components in execution traces, not the order in which they are executed in scenarios. These are also observed in our case studies.

Below, we provide a discussion of how REQRELEX compares with Salah’s [130, 95] and Egyed’s [60] approaches in the first experiment (first three rows for each case study).

First, we compare our results with Salah’s approach [130, 95] (rows 1 and 3 in Table 6.2 for each case study). We provide combined metrics that show how successful each approach is on finding a relationship between requirements without distinction on whether it is a dependency or a hint-relation. Based on the results of the case studies, REQRELEX performs better overall on finding the relationships before they are categorized as dependency or hint-relation on all case studies.

On finding hint-relations, REQRELEX performs very close or better on the case

studies that have higher number of hint-relations (Apache Pool [3] with 6, and Commons CLI [1] with 18), while it performs worse on the Chat System which has only 2 hint-relations. Upon investigation, we observed that REQRELEX categorizes those two hint-relations as dependencies because the features in both are stylistically used in a fixed order by the developers in test cases. Overall, however, REQRELEX achieves comparable or better results compared to Salah’s approach [130, 95] since it provides very close or better results on those case studies with a higher number of hint-relations. Overall, however, none of the approaches performs well on detecting hint-relations (compared to dependencies).

On finding dependencies (first three rows for each case study in Table 6.2), Salah’s approach [130, 95] performs worse on our case studies compared to both Egyed’s approach [60] and REQRELEX. Egyed’s approach [60] and REQRELEX perform very similarly on all case studies with small differences in their f-measures. Overall, REQRELEX achieves comparable or better results compared to Egyed’s approach [60], the state-of-the-art, in our case studies.

Next, we provide the results of another experiment (rows four and five for each case study in Table 6.2) we performed on the same case studies to show that REQRELEX is resistant to the selection of scenarios, unlike Egyed’s approach [60] (we do not provide results for Salah’s approach [130, 95] here, because there is no way to perform the experiment without fundamentally changing Salah’s algorithm since it depends on object flow). Scenario selection and gathering the execution traces of scenarios are very important steps that determine the success of dynamic analysis techniques. Since scenarios are typically created by developers manually, the process is open to mistakes. Therefore, it would be very beneficial for developers to use a technique that is somewhat resistant to such mistakes. In our experiment, we purposefully modified the collection of the execution traces in our scenarios for each requirement so that dependencies are

not explicit right in the execution traces. As an example, consider the two requirements `connect` and `sign-on` from the Chat System: `sign-on` has a dependency to `connect`. Therefore, when the components of the scenarios for these requirements are collected, the execution traces for `sign-on` should contain all components in the execution traces of `connect`. However, when developers create scenarios for requirements and collect execution traces of each scenario, they may only include the parts relevant to `sign-on` in its execution traces (purposefully or by mistake) and leave out the ones relating to `connect`. This is actually what is expected of developers during feature location as discussed in Section 4.2, which is the opposite of the expectation from them to find dependencies. For Egyed’s approach [60] to find the dependencies properly, all execution traces should be collected as described above (i.e. dependent requirements should explicitly contain the execution traces of their dependencies). For REQRELEX, however, this is not a necessity, because it doesn’t analyze the collected execution traces of scenarios to mine requirements dependencies. Instead, it uses those to find feature markers, and uses the test cases to mine the dependencies. Since REQRELEX uses probabilistic ranking on choosing feature markers, we argue that it will likely still choose good components to represent each feature, and be very resistant to such mistakes on scenario selection. Rows 4 and 5 in Table 6.2 for each case study show the same experiments run again to find requirements relationships. As the case study results suggest, REQRELEX obtains almost the same results, while Egyed’s approach [60] fails to find any of the dependencies in this new experiment, as expected.

Finally, Table 6.3 summarizes the applicability of each approach with pros and cons. Salah’s approach [130, 95] only works on object oriented systems since it relies on object flow and aliasing. On the other hand, Egyed’s approach [60] and REQRELEX work for all systems with a profiler available. Egyed’s approach [60] is very sensitive to the selection of scenarios, while REQRELEX is resistant to it. And finally, REQRELEX

Table 6.3. Comparison of the methods compared with REQRELEX

	Salah [130, 95]	Egyed [60]	REQRELEX
Requires scenarios	Yes	Yes	Yes
Works on object-oriented systems	Yes	Yes	Yes
Works on systems that are not object oriented	No	Yes	Yes
Resistant to scenario selection	—	No	Yes
Works in the absence of test cases	Yes	Yes	No

depends on the existence of test cases, while the other approaches do not.

In the rest of this section, we discuss advantages and limitations of our approach.

Our approach is independent from the programming language with which the system is built, unlike Salah’s approach [130, 95] which only works for object oriented systems. The only requirement of our approach is that it uses a profiler to gather execution traces, and the components in the execution trace are of the same type for tests and scenarios. Our tool currently works for Java [68], but it is easily extensible to work for any other language (such as object oriented, functional, procedural) for which a profiler is available (such as C, C++, python, perl, Smalltalk).

An advantage of our approach is that, even though it finds dependencies and hint-relations currently, it can easily be extended to find other types of relationships that can be deduced from test cases. We argue that test cases are a rich source of information that contain implicit knowledge about requirements relationships (such as dependencies and hint-relations as shown in this chapter). Therefore, they can be used for mining other types of requirements relationships.

Another big advantage of our approach is that, scenarios can be provided as test cases themselves. When the source code of the system is refactored or changed, developers fix tests to keep them passing after the changes. Since scenarios are also test

cases, developers will fix them too and scenarios will always stay up to date. Although this demands some effort from developers, the automatically mined requirements relationships will stay up to date as software evolves.

Another advantage of our approach is that it is resistant to the selection of scenarios. Scenario selection typically determines the success of dynamic analysis techniques, so it is beneficial for developers to use a method that is resistant to the selection of scenarios. This decreases the burden on developers by tolerating some mistakes.

Dependence on test cases might be listed as a disadvantage of our approach. However, although there may be some systems without test cases, we argue that it is commonplace for many production systems to have test cases to ensure correct behavior [107, 149]. Therefore we argue that our technique can still be successfully used for many production systems.

One limitation of our approach is how it uses test cases to find dependencies: If the developers of a test suite have a certain style such that they always exercise a requirement before another, even though there is no dependency between them, REQRELEX may wrongly deduce that there is a dependency. In fact, this was the reason that REQRELEX did not obtain 100% precision on the Chat System case study. This vulnerability can be fixed by using mutations to change the order of the exercised requirements in the test cases automatically, and checking if dependencies found are actual dependencies. We leave this as future work.

Another limitation of our approach is on finding requirements relationships overall. If the test suite does not contain test cases that exhibit the conceptual relationships (i.e. missing test cases), REQRELEX will miss them (hence the recall numbers are not 100% in our case studies). Similarly, the success of our approach is limited by the quality and properties of the test cases in the test suite. This is exhibited by the differences in the

results across different case studies. These can be partially mitigated by complementing REQRELEX with one of the existing techniques [60, 130, 95].

Another limitation of our approach is that it only applies to functional requirements currently. Our approach can be complemented with manual effort, as done in [60], to detect relationships for non-functional requirements (robustness, security), which we leave as future work.

Finally, our approach builds upon scenario based dynamic analysis techniques [119]. Therefore, it exhibits the same limitations for these techniques described in the related work section, such as missing coverage. This can be mitigated using multiple scenarios for each requirement as described in Section 4.4. to increase the coverage of dynamic analysis.

6.6.1 Threats to Validity

In this section, we discuss any issues that might have potentially affected our case study results and therefore may limit the interpretations and generalizations of our results.

The first threat is the number and type of the case studies we used and the extent they represent software systems used in practice. The Chat System is software used in a class at UCSD, and Apache Pool [3] and Apache Commons CLI [1] are open-source software commonly used in production. We picked our case studies from different domains to mitigate this threat. This threat can be reduced even further if more software systems of varying size from more domains are used for further experiments.

Another threat is the selection of functional requirements and scenarios to obtain execution traces for REQRELEX. We are not domain experts of the software used in the case studies. Therefore, we cannot claim that we found all requirements and our scenarios capture them best. Thus, depending on the chosen requirements and scenarios,

the results may differ.

Another threat is the preparation of the ground truth for requirements relationships in our case studies, which we performed manually. To mitigate risk, two different developers performed these tasks and the results are confirmed comparing their responses. However, it is still possible that mistakes have happened.

6.7 Conclusion

Requirements relationships describe different conceptual dependencies, constraints and associations between the requirements of a system [122].

Determining requirements relationships in a system is important on performing different tasks in the different lifecycle stages of the development of the system. During design and maintenance, requirements relationships help on determining possible requirements conflicts [60], and determining whether making a change on a requirement may have an impact on other requirements [27, 33, 91, 36, 124]. During maintenance, they help developers on program comprehension before a requirement is modified to help them understand the implications and investigate the related requirements that may help during the maintenance activity [153, 80, 117, 36]. During all stages, they help stakeholders see an overview of requirements to help with design and architecture maintenance decisions [143, 136, 131, 36].

The benefits of the identification of requirements relationships, described above, are possible only if the relationships are kept accurate and up-to-date. Acquiring and maintaining the relationships manually is error prone and time consuming [69, 99, 34]. Therefore it would be very beneficial to retrieve and maintain them automatically.

In this chapter, we propose retrieving and maintaining two types of requirements relationships automatically using existing test cases: dependencies and hint-relations. A requirement is dependent on another if it requires that requirement to be exercised before

itself to behave properly. There is a hint-relation between two requirements if they tend to be used together, but are not necessarily dependent on each other.

Testing is an important part of many software development processes, and a considerable amount of test code is produced in production systems based on empirical studies [107, 149]. We propose making use of this investment to automatically mine requirements relationships from existing test cases.

In this chapter, we build upon existing literature [60, 61, 119] to trace features in source code via scenarios. Once the features are traced in source code, we use highly relevant traces in the source code to find which test cases exercise which features. Finally, after identifying which test cases exercise which features, we perform statistical analysis to find relationships between requirements. We propose that if a requirement is always observed before another one, then there is a dependency of the latter requirement to the former. Similarly, if two requirements are observed in different orders in different test cases, but tend to be observed together many times, we then propose that they have a hint-relation.

Our approach achieves as good as or better precision, recall and f-measure values on the case studies we performed compared to the currently known approaches on finding requirements dependencies and hint-relations [60, 130, 95].

Our approach has many benefits: unlike existing methods [130, 95], it works for both object oriented systems and for any other type of systems with a profiler available. It is also resistant to the selection of scenarios, unlike existing techniques [60]. It is also fully automated with no need for human intervention during its analysis. The only requirements, similar to existing research [60, 130, 95], are to have a profiler available for the system to be analyzed and the creation of scenarios that represent requirements.

Finally, we propose an automated process and tool support, REQRELEX, to automatically find requirements relationships as a by-product of automated software

development processes such as continuous integration and Test-Driven-Development.

The work in this chapter, in full, is currently being prepared for submission for publication of the material. The dissertation author was the primary investigator and author of this material.

6.8 Future Work

If the test suite was developed stylistically to always exercise two requirements that do not depend on each other in the same order, our technique can be misguided and detect such requirements pairs to have a dependency. This can be mitigated by using controlled mutations to check if an automatically mined dependency is indeed a dependency or not.

Our technique currently only finds two types of relationships. It can be extended to make use of test cases to find more types of requirements relationships.

Chapter 7

Requirements Testing Progress: How Well Does the Test Suite of a System Cover Its Requirements?

In this chapter, we build upon the work described in Chapter 5 to find intents of test cases, and propose a new view that shows the progress of testing from the requirements perspective.

7.1 Introduction

Requirements Engineering extends across all steps in the development process, since requirements often change and new requirements come in. It is critical to identify and elicit requirements as early as possible in the software development lifecycle to minimize costs [30].

Once requirements are determined, the system is built in accordance with and to fulfill them. It is common practice to prioritize requirements, which typically drives the development and testing efforts [25, 140].

As development continues (or once it is completed), another important step of the software development lifecycle is testing. Testing uncovers bugs, demonstrates that the product works as expected and assures customers that the system conforms to

requirements. Based on the prioritization of requirements, it is natural to consider that features for those requirements deemed more important (as an example, features that are expected to be used more by the stakeholders) should be tested more than those deemed less important.

It is reported that the testing effort incurs approximately 40% —80% of the total development cost of the system [72]. If the testing phase is not monitored, more important features may be tested insufficiently; while testers may invest more time on less important features. This has a direct impact on the quality of the system. Therefore, it is critical to prioritize testing efforts according to the prioritization of the requirements [25, 140].

There are metrics that help monitor the progress of the testing phase, such as code coverage [110] and MC/DC (modified condition/decision) coverage [128]. These metrics demonstrate which parts of the system still remain to be executed by test cases, and how far off the testing phase is to completion from a source code perspective (i.e. to execution of all statements or decision paths in the source code). Although these are useful metrics, they provide information only on the source code level. They do not yield any information on the testing progress from a “requirements” perspective.

As motivation for our work, Figure 7.1 shows the number of test cases implemented by the developers for each requirement in Apache Pool [3]. Each bar in this chart corresponds to a requirement, and the height of the bar shows how many tests were implemented for a requirement. Given the information for Apache Pool [3], one reason for the difference in the number of tests for different requirements might be the priority of requirements. As an example, `borrow-object` and `return-object` have the most test cases, and inspecting the documentation of Apache Pool [3], these are the most important requirements. Therefore, developers might have implemented the most number of tests for these two requirements. On the other hand, with this reasoning, the requirements

Number of test cases in Apache Pool [3]

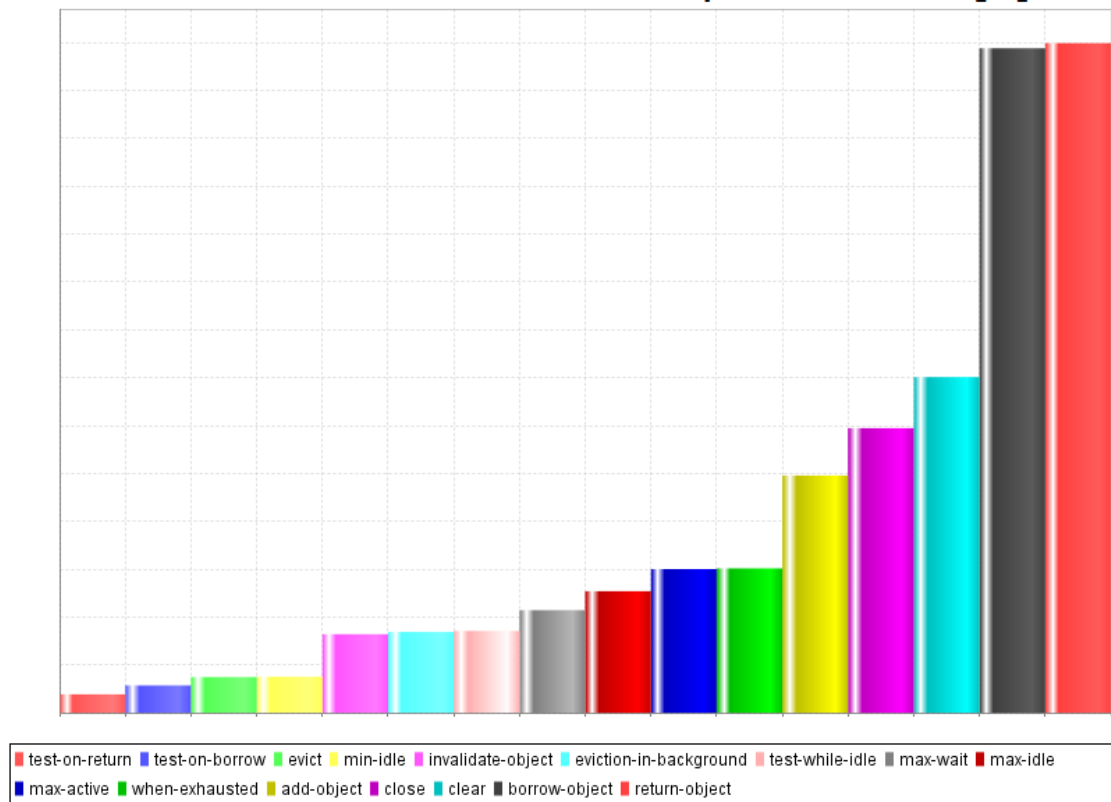


Figure 7.1. The number of test cases implemented by the developers for each requirement in Apache Pool [3]. Each bar in the chart corresponds to a requirement, and the height of the bar shows how many tests were implemented for a requirement.

towards the left of the chart seem to be the ones that are not as important as the ones towards the right. Therefore, the number of tests implemented for them is less.

However, investigating the number of bugs reported for Apache Pool [3], as shown in Figure 7.2, it turns out that some of the requirements that had less number of tests had more bugs reported about them. Importantly, the requirements `borrow-object` and `return-object` had a high number of bugs reported, which conforms to our initial reasoning that they were important requirements. Therefore, users of this library made heavy use of these requirements and they found bugs related to them. Furthermore, some of the requirements that had less number of tests have many reported bugs too. We

Number of bugs in Apache Pool [3]

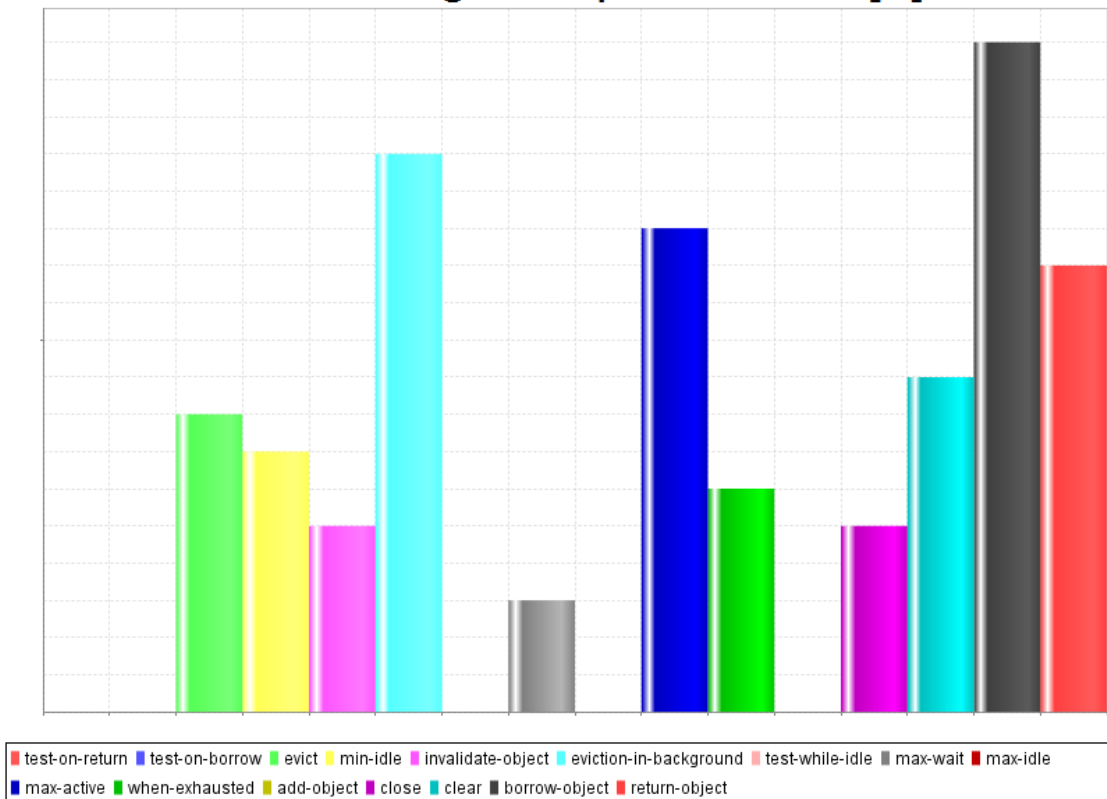


Figure 7.2. The number of bugs reported for each requirement in Apache Pool [3]. Each bar in the chart corresponds to a requirement, and the height of the bar shows how many bugs were reported for a requirement.

consider this as a sign that users of this library made heavy use of those requirements, and they found bugs for these requirements. Given that these requirements also had less tests, we consider this to be a natural outcome. Our work in this chapter provides a requirements level view over the testing phase to uncover the amount of testing performed for each requirement.

In this chapter, we propose a new view to demonstrate the testing progress on the requirements level. With this view, priorities of requirements can be associated with the effort put in testing each requirement. Based on this, stakeholders can understand which requirements were tested insufficiently, and allocate testing resources accordingly. We propose to answer the following questions with this view:

- What is the test suite focused on? Does it adequately test all requirements? How far off is the test suite from an ideal position with respect to testing all requirements adequately?
- If some requirements are more important than others, is the testing team investing the respective amount of effort for each requirement?

7.2 Contribution

To answer these questions, we make use of the RTM between requirements and test cases to estimate which requirement each test case targets to test (described in Chapter 5), and then use a weighted estimation algorithm to find the testing progress view, a view that shows the coverage of test cases over requirements. Our work makes the following contributions:

- A new view that shows the test suite's coverage of the requirements.

- A new method to automatically reverse engineer test suite coverage for requirements from existing test cases.
- A method to validate the proposed view.
- A tool to automate this process.

7.3 Related Work

For the purposes of the work in this chapter, we use the RTM between requirements and test cases, as exemplified in Table 2.3. In this RTM, rows are test cases, columns are requirements, and the ticks represent that a test case executes a requirement.

Our work in this chapter assumes that a high quality RTM exists for test cases. This can be obtained from the project if it is already maintained manually or via some tools by the stakeholders. In the case that it does not exist at all, one of the existing requirements traceability techniques in the literature [100, 60, 98] can be used to automatically reverse engineer it. These techniques trace requirements in test cases, i.e. they find which test cases execute which requirements. The work we describe in Chapter 4 is an example of these techniques.

Even though the literature on this subject is covered in detail in Section 4.2, we briefly review related literature here for completeness. In [100, 98], description of the requirements (as plain text) and the code of the test cases (again as plain text) are used. They assume that the documentation and code contain same or similar terms, and this can be exploited to find closeness between the requirement documentation and the test case. Given the description of a requirement and the test case code, i.e. two documents including plain text, terms are extracted from both using lexical analysis. Then the terms in both are compared to find how similar the two documents are. They propose that, if there are many similar terms in the two documents, the test case is said to execute the

requirement. Applying this for all requirements and test cases, an RTM as shown in Table 2.3 is obtained that shows which test cases execute which requirements. An advantage of these techniques is that since they rely on textual similarity, they can find requirements traces not only in test cases, but also in other software artifacts such as source code and architecture documents. Another advantage is that these techniques work for both functional and non-functional requirements. However, since these techniques rely on text similarity, they may have low accuracy (many false positives). This is due to the fact that naming conventions and documentation in software projects tend to get out of date as software evolves. Therefore, relying on textual similarity yields less accurate results over time. Another disadvantage of these techniques is that, requirements documentation may not have been existent at all, so it may be difficult to apply these techniques in the first place.

Another technique that works on finding requirements traces in test cases is described in Chapter 4. It uses scenarios, actions that trigger requirements, to execute functional requirements on the system. As scenarios execute, they profile the system to gather execution traces that contain information about the executed components (e.g. class and method names for object oriented systems). Then they find specific components that can represent each requirement, i.e. components that are observed in one requirement but no others. Once these specific components are found for each requirement, test cases are executed and profiled similar to scenarios. If the execution trace of a test case contains the specific component chosen for a requirement, that test case is said to execute that requirement. Performing this on all requirements and test cases, they find an RTM as shown in Table 2.3. These approaches require creating scenarios for requirements, which means extra work for developers. However, they yield more accurate results compared to the previous family of approaches, since they use a more accurate representation of requirements: an executable one compared to a textual description. These approaches

do not require any documentation for requirements. A disadvantage of these approaches is that they only work for functional requirements. These approaches perform dynamic analysis, so they may miss the tracing links on the parts of the system that are not executed by the scenarios.

Once the RTM is captured from the project, or by using one of the techniques in the literature as described above [100, 98], our technique can automatically reverse engineer the testing progress from existing test cases.

Although not directly related to our work, there has been extensive research on finding testing progress of a software system from different perspectives.

An S-curve [83] is used to monitor the progress of the execution of a test plan. As discussed in the introduction section, it works with the premise that there is a test plan, and it shows how close the testing efforts are to the actual test plan, along with how many of the implemented tests pass and fail, and whether there is a delay in the testing efforts. This can be used in the existence of a test plan produced before starting the testing phase. However, this cannot be used in projects without a test plan.

Many other metrics exist to monitor the success of the testing efforts: defect arrivals over time, defect backlog over time, test confidence, test efficiency [58], code coverage [110], MC/DC coverage [128] and many more. Weyuker [146] analyzed some of these metrics to discuss their adequacy to determine whether or not sufficient testing has been performed. These metrics monitor the testing efforts from a success standpoint, i.e. how much they help increasing the quality of the software built. While these are all effective software testing metrics, they do not provide progress information from a requirements perspective.

Although unrelated to this work, there is existing literature on extracting requirements level views for different aspects of a system. In [151], requirements are associated with their overall scattering across source code components. In [85], a similarity metric

is proposed to demonstrate how similar requirements are based on the closeness of their implementations. These are not directly related to our work, but we list them here since they also provide requirements level views on different aspects of software systems.

7.4 REQTESTPRO: Requirements Level View for Testing Progress

REQTESTPRO —Requirements Level View for Testing Progress —is our tool that automates the process of obtaining testing progress views from existing test cases on the requirements level. Figure 7.3 summarizes the inputs, steps and outputs of REQTESTPRO.

The inputs are the RTM and numeric priorities for each requirement. Based on the RTM, we first calculate a **weighted-RTM** where more weight is assigned to a requirement in a test case if the test case targets to test that requirement.

Once all requirements are weighed for all test cases, we sum the total weight for a requirement across all test cases, and divide it to the priority of the requirement. This yields a “progress score” on how well that requirement has been covered by the test suite.

Finally, we display the progress scores for each requirement in an easy to understand format.

In the rest of this section, we discuss each step in detail.

7.4.1 Step 1: Inputs —RTM and Requirement Priorities

As discussed in the earlier sections, our technique works under the assumption that an RTM between requirements and test cases exists, or one can be obtained using one of the techniques in the literature [100, 98].

We also take, as input, priorities for each requirement. These are simply numbers that denote the importance of requirements with respect to each other. A higher number means a more important requirement. To assign importance to requirements, existing

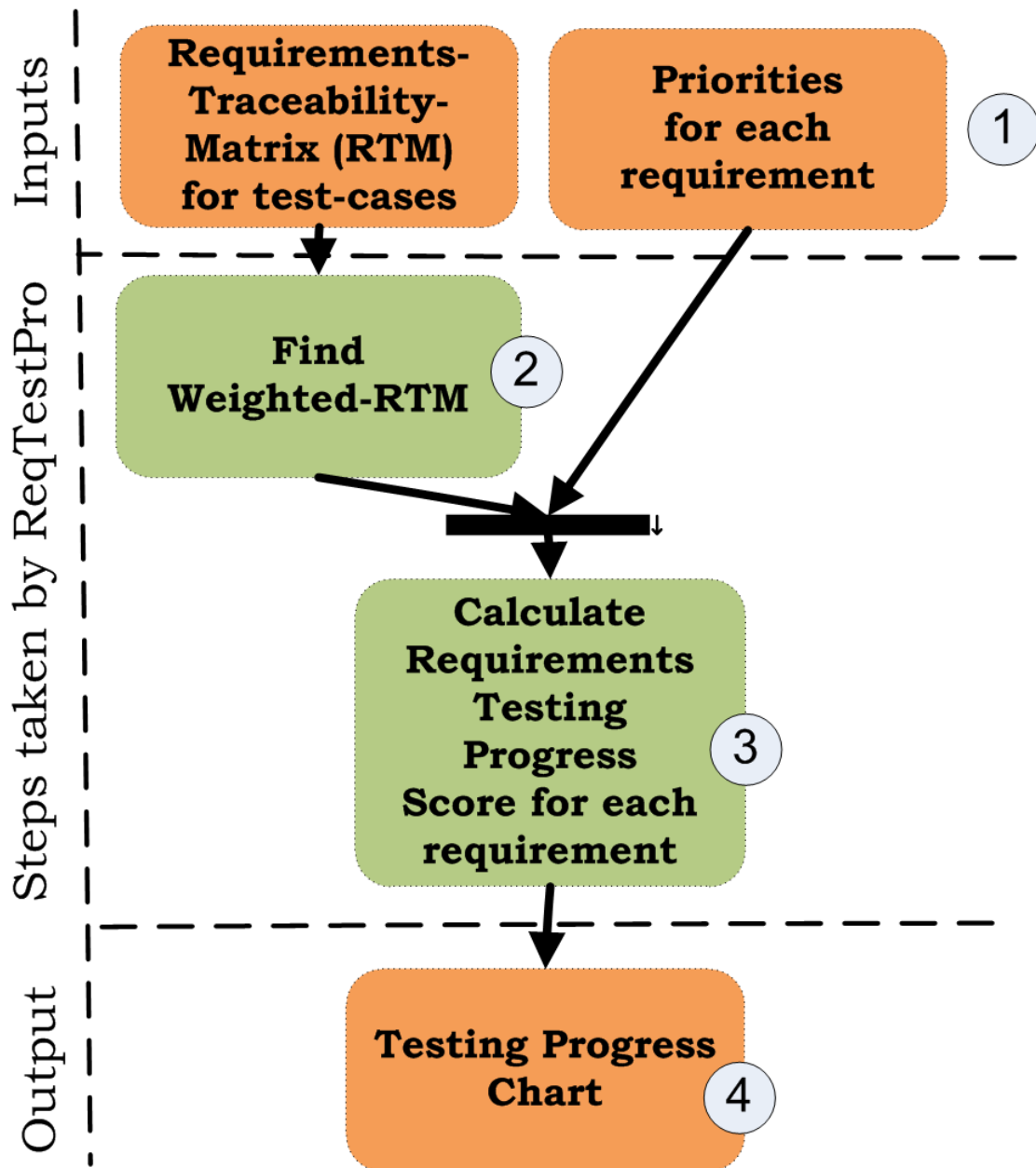


Figure 7.3. Inputs, steps and outputs of REQTESTPRO. The inputs are the RTM and requirement priorities. The output is the testing progress view over requirements.

Table 7.1. Priorities assigned to each requirement in the Chat System. A higher number corresponds to a higher priority, i.e. a more important requirement.

Requirement	Priority
connect	1.0
sign-on	1.0
send-message	1.0
sign-off	0.5

Table 7.2. Sample Requirements Traceability Matrix for the tests of the Chat System (duplicate of Table 2.3 for convenience).

	connect	sign-on	send-message	sign-off
t_1 : testConnect	✓			
t_2 : testConnectAndSignOn	✓	✓		
t_3 : testSendMessage	✓	✓	✓	
t_4 : testSignOff	✓	✓		✓

techniques in the literature can be used [40, 84, 129, 24]. For our running example, the priorities we assigned for each requirement are shown in Table 7.1. A higher number corresponds to a higher priority, i.e. a more important requirement.

7.4.2 Step 2: Find Weighted-RTM

Given an RTM, we make the following observation: test cases are typically implemented to test and confirm that requirements work as expected. For this reason, a test case t would be expected to execute a requirement r if it's testing r , and would not execute any other requirements unless they are required to be executed to be able to test the behavior of r . Otherwise, a test case would contain redundant code, would be less readable, and developers would need to invest more time on implementing the test case.

As an example, consider the Chat System: one of the tests for the Chat System is `testSendMessage` (see Table 7.2). As its name suggests, it is implemented to test the behavior of the requirement `send-message`. Based on the RTM (see Table 7.2), however, it also executes the requirements `connect` and `sign-on`. This is because those two

requirements need to be executed first to be able to test `send-message` (`send-message` depends on `sign-on`, `sign-on` depends on `connect`). Furthermore, `testSendMessage` does not execute `sign-off`, because it does not need to, to test `send-message`.

Based on this observation, we propose that in the test `testSendMessage`, since the real target is `send-message`, `testSendMessage` should be considered a test case that has more weight on testing `send-message`, and less weight on `sign-on` and `connect` (still non-zero weights though since it still executes them). This step builds upon our previous work on finding test intents described in Chapter 5 (finding the requirement that a test case is aiming to test). However, the idea here is not to find a single requirement for a test case as discussed in Chapter 5, but rather to assign weights to each requirement executed by a test case.

We proceed in two steps:

1. Rank all requirements for a test case based on how likely each requirement is to be the target;
2. Assign higher weights to more likely requirements, and lower weights to the less likely ones.

For the ranking, we use Formal Concept Analysis [66], similar to what we have done in Sections 5.4.1 and 5.4.2.

To assign weights, we use test cases as objects and requirements as attributes of a formal context (see Tables 7.2 and 7.3). Then we find the concept lattice out of the formal context, as shown in Figure 7.4. Finally, we perform a bottom-up traversal of the lattice for each test case to rank the requirements. As an example, consider `testSendMessage` again. If we perform a bottom-up traversal for it, the order in which we iterate the attributes would be: `send-message`, `sign-on`, `connect`.

We would expect that FCA will perform well on ranking the requirements in the

Table 7.3. Concepts in the formal context for the Chat System RTM given in Table 7.2

Concept	(objects, attributes)
c_1	$(\{t_1, t_2, t_3, t_4\}, \{\text{connect}\})$
c_2	$(\{t_2, t_3, t_4\}, \{\text{connect}, \text{sign-on}\})$
c_3	$(\{t_4\}, \{\text{connect}, \text{sign-on}, \text{sign-off}\})$
c_4	$(\{t_3\}, \{\text{connect}, \text{sign-on}, \text{send-message}\})$
c_5	$(\emptyset, \{\text{connect}, \text{sign-on}, \text{send-message}, \text{sign-off}\})$

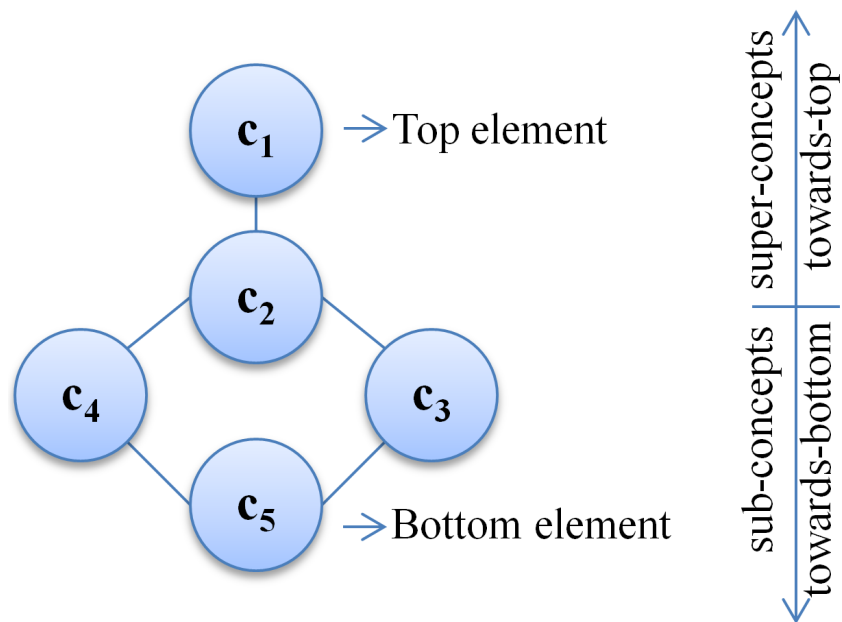
**Figure 7.4.** Concept lattice for the formal context shown in Table 7.3.

Table 7.4. Weighted-RTM for the RTM given in Table 7.2 (geometric sequence base = 10)

	connect	sign-on	send-message	sign-off
t_1 : testConnect	1.00			
t_2 : testConnectAndSignOn	0.09	0.91		
t_3 : testSendMessage	0.01	0.09	0.90	
t_4 : testSignOff	0.01	0.09		0.90

presence of a decent quality test suite. However, if the quality of the test suite is low, then FCA will not perform as good, because it will be more likely to make mistakes during ranking.

Once the ranking is performed, we assign weights to each requirement executed by a test case. Considering our Chat System example, the requirements executed by testSendMessage were ranked, from more important to less important, as: send-message, sign-on, connect. Assuming that we use a geometric sequence (with the geometric sequence base as 10, and the total sum of the sequence to be equal to 1) to assign weights to requirements, the weights assigned would be:

connect : 0.01

sign-on : 0.09

send-message : 0.90

We perform a similar weighting for all test cases and requirements, and we call the resulting RTM that has weights assigned a weighted-RTM. Table 7.4 shows the weighted-RTM for the RTM shown in Table 7.2 (the base of the geometric sequence is 10).

7.4.3 Step 3: Calculating Progress for Each Requirement

Once we have the weighted-RTM, we calculate a progress score for each requirement. This score shows how far off the testing of the requirement is, based on its

Table 7.5. Notation used to calculate the progress scores in REQTESTPRO.

<u>Symbol</u>	<u>Description</u>
m	total number of requirements
n	total number of test cases
r_j	the j^{th} requirement
p_j	the priority (importance) of r_j
t_i	the i^{th} test case
$R = \{r_j \mid 1 \leq j \leq m\}$	the set of all requirements
w_{ij}	the weight assigned to r_j in t_i
P_j	progress score for r_j

priority.

For a system that has m requirements and n test cases, consider the notation in Table 7.5. The progress score for a requirement is calculated by finding the total weights assigned to that requirement across all test cases, and dividing the total to the priority of the requirement. Formally, progress for a requirement (P_j) is defined as follows:

$$P_j = \frac{I}{n \times p_j} \times \sum_{i=1}^n w_{ij} \quad (\forall j : r_j \in R) \quad (7.1)$$

where:

$$I = \sum_{j=1}^m p_j$$

P_j represents the progress of the testing of requirement j compared to the other requirements. As an example, consider the priorities assigned to the requirements for the Chat System shown in Table 7.1. In our example Chat System, $I = 4$. And:

$$P_{\text{connect}} = \frac{4}{4 \times 1} \times (1.00 + 0.09 + 0.01 + 0.01) = 1.11$$

Similarly, the progress score for each requirement would be:

$$\begin{aligned} P &= [P_{\text{connect}}, P_{\text{sign-on}}, P_{\text{send-message}}, P_{\text{sign-off}}] \\ &= [1.11, 1.09, 0.90, 1.80] \end{aligned}$$

These scores summarize the test suite with the following reasoning:

- There is 1 test for each requirement (as observed in the names of the tests). Therefore, the resulting progress scores should be close to 1 for those requirements with a priority value of 1, and higher for those with a priority value less than 1.
- Even though each requirement has 1 test case, requirements executed more times than others are assigned a slightly higher weight. As an example, `connect` is executed in all test cases, so it has the highest score; while `send-message` and `sign-off` were executed the least, hence have the lowest scores.
- The progress score of `sign-off` is higher than the others, because it was assigned a lower priority, yet it has the same number of tests for it as the other requirements.

7.4.4 Step 4: Output

As discussed in the previous subsection, the output of REQTESTPRO is an m -dimensional vector that shows the progress score for each requirement. For the Chat System, the output would be:

$$P = [1.11, 1.09, 0.90, 1.80]$$

For easier comprehension, we visualize this as a bar chart. The chart shows the respective progress for each requirement compared to the expected baseline. Figure 7.5 shows such a sample output for our Chat System example.

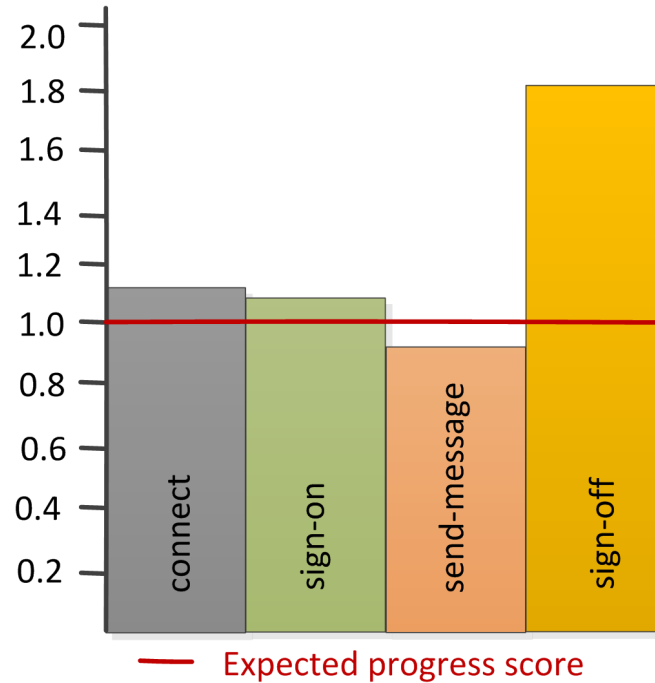


Figure 7.5. Testing Progress visualized as a bar chart.

Here, the progress of each requirement is charted along with the baseline. Ideally, all requirements should have a progress score equal to the baseline. Those that are above have been tested more than needed, while those that are under have been tested insufficiently.

7.5 Evaluation

In this section, we evaluate the usefulness of the proposed testing progress view on case studies. We also discuss how to choose the parameters of our technique to yield good results.

7.5.1 Case Studies

To evaluate our approach, we used three of our case studies discussed in Section 2.3: UCSD Chat System, Apache Pool [3] and Apache Commons CLI [1].

Table 7.6. Properties of the case studies used in the evaluation of REQTESTPRO.

Case Study	# Requirements	# Test Cases
UCSD Chat System	16	20
Apache Pool [3]	16	77
Apache Commons CLI [1]	11	181

Table 7.6 lists statistics for each case study relevant to our evaluation. For all of the case studies, we identified the requirements manually from their documentation, source code and comments as discussed in Section 2.3. Similarly, we manually identified which of the test cases in their test suites are implemented to test those requirements.

7.5.2 Evaluation Criteria

There exist metrics such as code coverage [110] and MC/DC coverage [128] to monitor the progress of testing on the source code level. However, we propose a metric to measure testing progress on the requirements level. Therefore, we evaluated our approach by comparing the progress scores found by REQTESTPRO to the progress scores obtained using ground truth that we prepared for each case study manually.

We prepared the ground truths for our case studies by manually inspecting the test cases and identifying which requirements they test. When a test case tests the interaction of two or more requirements, we listed them all and assigned equal weights to all of them.

To evaluate our approach, we compare the progress score vector found by REQTESTPRO with the progress score vector found on the ground truth for each case study manually. We prepared the ground truths for our case studies by manually inspecting the test cases and identifying which requirements they test. When a test case tests the interaction of two or more requirements, we listed them all and assigned equal weights

to all of them. As an example, the ground truth for the running example is:

$$P_{truth} = [1, 1, 1, 2]$$

since there is a single test case for each requirement (as one can observe looking at test names). To evaluate our approach, we compare the progress score vector found by REQTESTPRO with the progress score vector manually found on the ground truth for each case study.

7.5.3 Evaluation Results

For evaluation purposes, we needed to devise a distance function between two m-dimensional progress score vectors that would geometrically make sense.

A commonly used similarity metric is cosine similarity [138]. Given two vectors i and j of the same size:

$$similarity(i, j) = \cos(\theta) = \frac{i \cdot j}{\|i\| \times \|j\|} \quad (7.2)$$

where θ is the angle between the given vectors, and $\|i\|$ is the L_2 norm of the vector (i.e. its length). Based on cosine similarity, angular distance is defined as:

$$angular-distance(i, j) = 1 - \frac{\cos^{-1}(similarity(i, j))}{\pi} \quad (7.3)$$

Note that angular distance is a proper distance metric [92]. In our case, using angular distance as the distance function makes sense, because progress scores represent the progress score magnitudes in each direction in the m-dimensional space, and angular distance measures how different the two vectors are in their angular direction. If two vectors are roughly in the same direction, the test progress scores are close and their

Table 7.7. Percentage distances of progress scores between those found by REQTESTPRO and the ground truth*

Case Study	% Distance to Ground Truth
UCSD Chat System	8.94%
Apache Pool [3]	26.32%
Apache Commons CLI [1]	25.73%

* all requirements are assumed to have the same priority, and base of the geometric sequence is 10

angular distance is small.

In the rest of this chapter, when we use distance, we mean the angular distance between what REQTESTPRO finds and the ground truth.

Table 7.7 shows the results of REQTESTPRO on the case studies, where all requirements are assumed to have equal priority, and a geometric sequence with base 10 is used. The values shown are the angular distances of the testing progress scores found by REQTESTPRO and the ground truth. The results suggest that REQTESTPRO finds results within 8.94% —26.32% of ground truth.

Below are listed important points to be considered during our evaluation:

During our experiments, we use geometric sequences to assign weights to requirements for each test case (as discussed in Section 7.4.3). Whether using different bases impacts the results, and why should be justified.

The distance function used for evaluation should geometrically make sense, and ideally be independent of requirement priorities. This would make the results listed in Table 7.7 independent of priorities used in the experiment.

Below, we ask evaluation questions (EQ) related to these and answer them.

EQ 1: Is the chosen distance function independent of priorities (importance) assigned to requirements?

Having a distance function that is independent of assigned priorities would be ideal. This would make sure the results of our experiments can be trusted even under changes to priorities we assigned to requirements during our experiments. Angular distance, however, does not have this property. Although geometrically it is intuitive to find the distances in our experiments using it, it is dependent on the priorities. We also considered other commonly used distance functions, such as Euclidean distance [53], Mahalanobis distance [103], and Chebyshev distance [37]. However, none of these distance functions were independent of priorities neither. Furthermore, they did not have a bounded range (see **EQ 2**). Therefore, we opted to use angular distance, even though it is dependent on priorities. However, to mitigate this factor, we asked another evaluation question later on (see **EQ 3**) and performed experiments to answer it. There are other candidate distance functions to be considered, which we leave as future work.

EQ 2: Does the distance function have a pre-defined upper and lower bound?

It is important to perceive the distance between the testing progress scores of a case study and its ground truth (such as % difference). If the distance is not bounded, then there is no way to analyze what a found distance value means compared to the baseline. Angular distance does have a range: $[0, \pi]$.

EQ 3: How dependent is the distance function on the requirements priorities? Does it exhibit intolerable variations based on differing priorities?

Since the answer to **EQ 1** is no, we wanted to test whether the distances found using angular distance showed much variation under different priorities. We do this by regression, i.e. experiments performed with random priorities many times. If the variation between the distances is small within a case study (compared to the average

Table 7.8. Minimum, maximum and average percentage distances of progress scores between those found by REQTESTPRO and the ground truth in 100.000 regression runs.

Case Study	Minimum % Distance	Maximum % Distance	Average % Distance
UCSD Chat System	2.47%	24.77%	8.89%
Apache Pool [3]	10.45%	37.25%	24.77%
Apache Commons CLI [1]	11.28%	32.15%	23.81%

and the ones reported in Table 7.7), we can propose that the results we present in this chapter can be relied upon.

For each case study, we performed 100.000 runs of the experiment with priorities randomly chosen from 1 to 5. This is based on the prioritization scheme suggested in RFC 2119 [32] and IEEE Std. 830-1998 [41], and in line with reported common usage in practice [134, 93] (note that, REQTESTPRO supports any other types of priority assignments as long as the priorities can be denoted as numbers).

Table 7.8 shows the minimum and maximum percent distances of what REQTESTPRO finds and the ground truth at the end of 100.000 runs. The results suggest that REQTESTPRO is within 27.18% —37.25% of the ground truth in the worst case, and within 8.89% —24.77% on average. Furthermore, results for each case study suggest that even though angular distance is dependent on requirements priorities, the range of distances between what REQTESTPRO finds and the ground truths vary by 26.8% at the most (in Apache Pool [3]), considering the difference between minimum and maximum distances. Therefore, we propose that angular distance can be used as a distance function for our purposes tolerably.

EQ 4: How much difference does the use of geometric sequences with different bases make on assigning weights to requirements?

When the base of a geometric sequence is larger, the difference between two consecutive weight assignments will be higher. Using a larger geometric base vs. a

lower one would make a difference for those case studies where ranking works better. As discussed in section 7.4.2, finding a good weighted-RTM depends on how well FCA works on a case study. Therefore, it depends on the characteristics of the test suite.

Overall, if FCA works well for a case-study, using a higher geometric base would result in a lower distance to ground truth; because this means assigning very high weights to those requirements found by FCA to be more specific to a test case. This, in turn, brings the automatically found weighted-RTM closer to the actual weighted-RTM (the ground truth); hence the distance decreases. In summary, using a high geometric base would improve results for a case study on which FCA performs well.

On the other hand, if FCA does not perform well on a case study, using a high geometric base would reinforce the performance of FCA on the weights assigned in the weighted-RTM. This, in turn, would drift the weighted-RTM found by REQTESTPRO away from the ground truth; hence the distances would be expected to be higher for higher geometric bases; while lower geometric bases would distribute the error margin of FCA across found weights, and keep the weighted-RTM found as close as possible to the ground truth. Therefore, the distance would be expected to be less. In summary, using a high geometric base would worsen results for a case study on which FCA does not perform well.

To confirm the idea proposed above, we investigated the case study results using equal priorities for each requirement (reported in Table 7.8), with varying geometric bases on assigning weights. Table 7.9 lists the results of the experiment reported in Table 7.7 with the same parameters, except with different geometric bases (in Table 7.7, the base is fixed as 10).

The results in Table 7.9 confirm the hypothesis discussed above. First, based on Table 7.7, REQTESTPRO performs better on the Chat System, while it performs worse on Apache Pool [3] and Apache Commons CLI [1]. And conforming to the hypothesis

Table 7.9. Percentage distances of progress scores between those found by REQTESTPRO and the ground truth under varying geometric bases*.

Geometric Base	% Distance to Ground Truth		
	UCSD Chat System	Apache Pool [3]	Apache Commons CLI [1]
2	21.61%	24.56%	20.80%
3	15.57%	25.26%	22.10%
4	12.62%	25.60%	23.17%
5	11.06%	25.82%	23.93%
6	10.18%	25.98%	24.50%
7	9.65%	26.10%	24.92%
8	9.31%	26.19%	25.25%
9	9.09%	26.26%	25.51%
10	8.94%	26.32%	25.73%
...
20	8.54%	26.62%	26.74%

* all requirements are assumed to have the same priority

discussed above, higher geometric bases yield better results for the Chat System, while lower geometric bases yield better results for Apache Pool [3] and Apache Commons CLI [1].

Overall, however, we propose using 10 as the geometric base considering that the results for Apache Pool [3] and Apache Commons CLI [1] are not much worse than using 2; while for the Chat System (where FCA performs well), the difference is more subtle.

Finally, it is worth noting what happens if the geometric base is further increased. Using geometric base as 20, the distances are: 8.54% for the Chat System, 26.62% for Apache Pool [3], and 26.74% for Apache Commons CLI [1]. This demonstrates that the results converge very quickly for further increasing base values. This is expected since the weights distributed using geometric sequences change very slightly once the base is larger than a certain threshold.

7.6 Discussion

In this chapter, we proposed a technique that takes in the RTM between requirements and test cases as input, and automatically finds testing progress views in two steps:

- Automatically rank requirements for each test case, and assign weights to each requirement based on whether a requirement is the main target of a test case. This yields the weighted-RTM.
- Use the weighted-RTM to automatically find testing progress over requirements according to the respective importance of requirements.

In the evaluation section, we compared the testing progress results found by REQTESTPRO using the angular distance as the distance function. The evaluation questions discussed and justified the use and choice of the parameters and the distance function used in evaluation.

Based on our case study results, our proposed view is useful to estimate the testing progress over requirements, and it can be useful in production systems. With this view, stakeholders can see:

- what the test suite is focused on from a requirements perspective.
- if the test suite adequately tests all requirements.
- how far off the test suite is from an ideal position with respect to testing all requirements adequately.

To further motivate the usefulness of the testing progress view, Figure 7.6 shows the testing progress of Apache Pool that REQTESTPRO outputs along with the number of bugs reported for each requirement in Apache Pool [3]. To obtain the testing

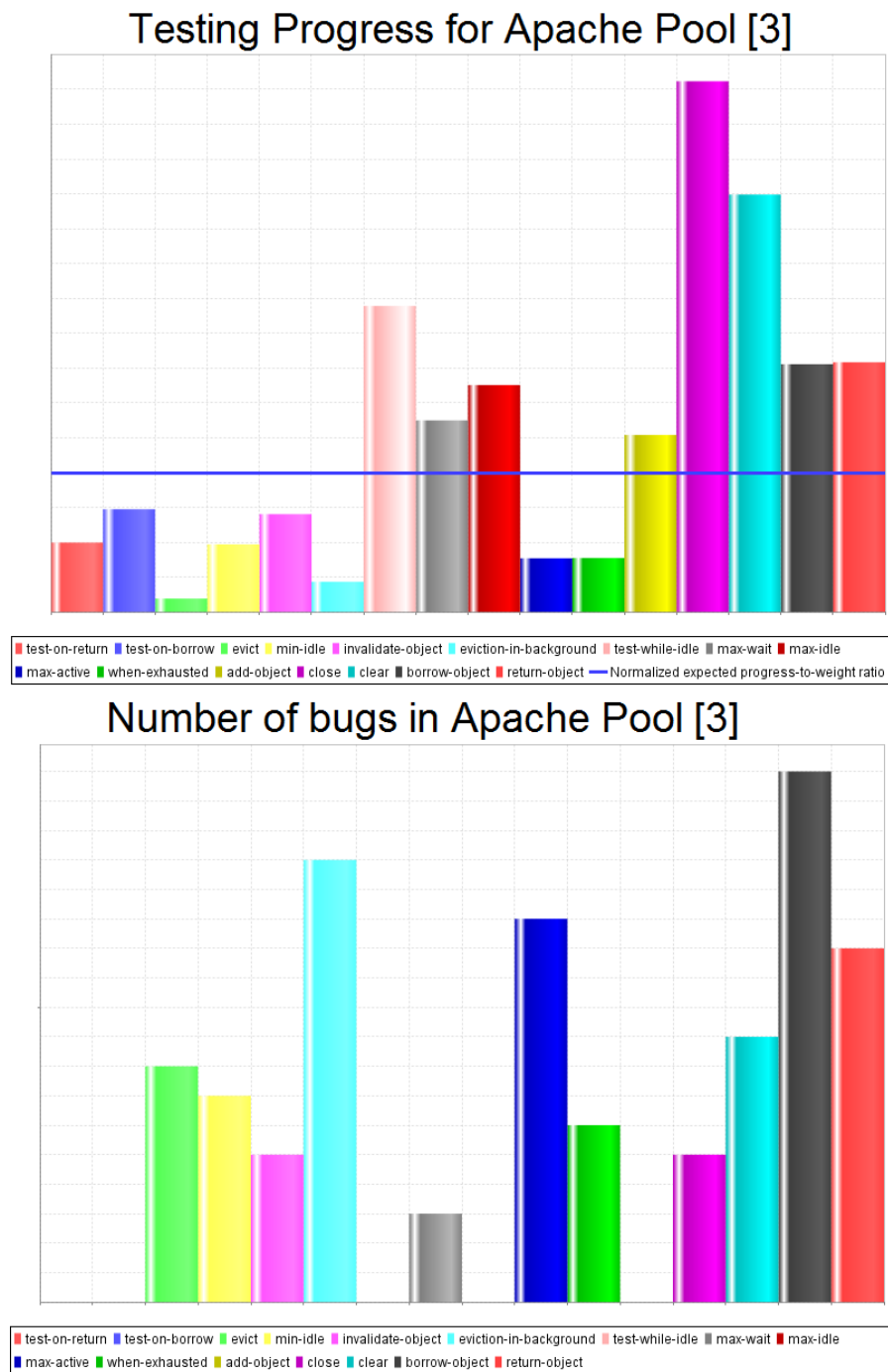


Figure 7.6. The testing progress of Apache Pool that REQTESTPRO outputs along with the number of bugs reported for each requirement in Apache Pool [3]. Many requirements that are considered to have inadequate testing by REQTESTPRO (the ones under the line at the top chart) have a higher number of bugs reported.

progress view for Apache Pool [3], we consulted a domain expert to assign priorities to requirements, based on which the progress view is obtained. Based on this figure, many requirements that are considered to have inadequate testing by REQTESTPRO (evict, min-idle, invalidate-object, eviction-in-background, max-active, when-exhausted) have a higher number of bugs reported. Considering this output, we propose that the testing progress view can be a useful measure of where the testing efforts should have been invested. This way, the number of bugs for those requirements could be reduced.

In the rest of this section, we discuss advantages and disadvantages of our approach.

REQTESTPRO provides a requirements level view of the testing progress, unlike source code level metrics. This empowers stakeholders on making prioritization decisions as the testing phase continues, and provides a high level view over the testing efforts.

Our approach assumes the existence of an RTM between requirements and test cases. Such an RTM might already exist in some systems. Otherwise, it can be obtained by using one of the techniques that exist in the literature [100, 98].

REQTESTPRO works for both functional and non-functional requirements, as long as both types of requirements exist in the RTM.

REQTESTPRO is independent of the programming language the system is built on. The only requirement is the existence of the RTM on test cases.

REQTESTPRO supports assigning different priorities for requirements. This allows having a testing progress view that can support maintenance, management and prioritization decisions on the testing phase.

REQTESTPRO depends on the existence of a test suite. It can only be used for those projects that have test cases already. However, since its aim is to provide a testing progress view, this is expected.

The performance of our approach is dependent on the quality of the test suite. As our case study results suggest, it yields good results on software with a high quality test suite (the Chat System case study). It also provides results within a limited error margin for other systems. As the results of the Apache Pool [3] and Apache Commons CLI [1] suggest, even though their test suites are not as good a fit for our approach as the chat system, REQTESTPRO still provides a testing progress view that has limited deviation, and would be useful to stakeholders.

A factor that impacts the performance of REQTESTPRO, other than the quality of the test suite, is the properties of the requirements of the system. Based on our investigations on why the Chat System results are better than the other two case studies, we observed that the Chat System requirements mostly have dependencies between each other. Therefore, its test cases execute requirements only if they are required (our initial observation based on which we proposed to use FCA). On such a case study, FCA performs well. However, there are test cases that test the interaction of requirements (which are not dependent on each other) in the other case studies. Ideally, when the interaction of requirements is tested, they should be assigned equal weights in the weighted-RTM. However, FCA ranks requirements and assigns decreasing weights to them from more specific to less specific. This results in mis-weighting some requirements in the weighted-RTM. Therefore, the found testing progress can get worse than expected. This issue can be mitigated by using the approach discussed in Chapter 5 and existing techniques in the literature [60, 130, 95] to explicitly find dependencies between requirements, and use this information during the assignments of weights in the weighted-RTM (instead of geometric sequences across the board for all requirements). We leave this as future work.

7.6.1 Threats to Validity

In this section, we discuss any issues that might have potentially affected our case study results and therefore may limit the interpretations and generalizations of our results.

First, we cannot claim that the case studies we used represent the full extent of production systems in practice. We chose the case studies from different domains to mitigate this threat. It can be further reduced if more case studies from different domains are experimented with.

Second, since we are not domain experts of software in our case studies, we cannot claim that we identified all requirements for each of them. Furthermore, we cannot claim that the ground truths we prepared are completely correct. We asked two developers to independently perform these tasks and confirmed the results. However, mistakes might still have happened.

Finally, we cannot claim that the distance function we use in the evaluation section is the best one. We analyzed its properties and mitigated any threats it poses for the reliability of our results in the evaluation section. However, experimenting with other distance functions to confirm our results would further decrease this risk, which we leave as future work.

7.7 Conclusion

Requirements Engineering (RE) is an important step in the software development lifecycle. Requirements are typically prioritized in RE stage, which drives development and testing efforts.

Testing is another important step in the software development lifecycle. Many, if not all, software teams perform testing to make sure the system conforms to the

requirements, and behaves as expected. However, testing is a costly activity [149] that can consume 40% —80% of the total budget of the software development effort [72].

Unless the testing phase is monitored, testers may test high priority requirements insufficiently, and less important ones more than required. This directly impacts the quality of the system. Therefore, it is critical to prioritize testing efforts according to the prioritization of the requirements [25, 140].

In this chapter, we propose a new view that shows the test suite’s coverage of the requirements. We propose that, using this view, stakeholders can monitor the testing phase to make sure requirements are tested in accordance to their priorities.

Our tool REQTESTPRO automatically reverse engineers test suite coverage for requirements from existing test cases. We also provide a method to evaluate the effectiveness of our proposed view, and discuss that the evaluation criteria is sound and reliable. We assess our technique on three case studies: a Chat System, Apache Pool [3] and Apache Commons CLI [1]. On these case studies, REQTESTPRO obtains results within 75.23% —91.11% close to the baseline results on average, using the evaluation method we propose.

Finally, we provide tool support that completely automates this process.

The work in this chapter, in full, is a reprint of the material as it appears in: “Celal Ziftci and Ingolf Krüger. Getting More From Requirements Traceability: Requirements Testing Progress. In *Traceability in Emerging Forms of Software Engineering*, pages 12-18, San Francisco, California, USA, 2013. IEEE.” The dissertation author was the primary investigator and author of this paper.

7.8 Future Work

As explained in the discussion section, REQTESTPRO can perform better if requirements dependencies and interactions are found and used on calculating the weighted-

RTM.

Another research area is to assign importance to test cases automatically and use that information during the calculation of the weighted-RTM. Currently, we assume all test cases are equally important. However, it may be the case that some test cases have higher importance in a test suite: they might demonstrate critical behavior, or they might be testing an important use case. Using such information would improve the testing progress metric.

Another research direction is providing testing progress views for requirements grouped into higher levels. This is especially important for systems where the requirements don't map to features on-to-one. In such systems, multiple features might map to a single requirement. If grouping is supported in REQTESTPRO, the testing progress view would provide more value for such systems. Requirements may be ordered in a hierarchy where each lists more detailed requirements. The testing progress metric proposed in this work can be accommodated to encapsulate requirements into groups and provide testing progress for different levels in the hierarchy.

A field that might benefit REQTESTPRO is 'code clone detection', where the target is finding locations in code where a piece of source code was copied and pasted, hence duplicated. If code cloning is detected in a certain part of the source code, the coverage view generated by REQTESTPRO can make use of this information to adjust the coverage of the requirement tested.

Another important area of further improvement on REQTESTPRO would be to use more information on the 'contents' of each test case. REQTESTPRO currently does not consider the following information:

- How many times does a test exercise a feature? This may be important when a test case exercises a feature in many different ways, such as in a loop.

- Does a test exercise a requirement exhaustively? As an example, a test might be testing a feature to make sure the boundary cases are exhausted.

Adding these information would make REQTESTPRO more effective in determining the testing progress.

Chapter 8

Conclusion and Future Outlook

In this chapter, we conclude the work in this dissertation and propose future work that opens up the avenue for future research in this field.

8.1 Conclusion

This dissertation focused on mining test cases to aid software maintenance tasks. We form a convection cycle between requirements and tests to help stakeholders understand relationships between requirements, and their relationships with tests.

We first built upon feature location techniques in the literature to relax some of the assumptions of the existing techniques. Our results suggest that feature location can be performed as a repeatable act and can be performed using the existing tests of the system.

Then, we used our feature location technique to find traceability links between requirements and test cases. Our results suggest that, using feature location, the accuracy of traceability improves compared to existing methods.

The work in these chapters formed the foundation of the rest of the results in this dissertation. Building upon our traceability approach, we first demonstrated a new technique to enhance the semantics of requirements trace links via test intents. Our results suggest that our technique can provide test intents to stakeholders successfully.

Then we proposed a new technique to mine different types of relationships between requirements. Our results suggest that our approach can yield comparable or better results than state of the art.

Finally, we proposed a new requirements level view over the testing phase to show where the testing efforts are invested, and whether this is in line with the importance of requirements.

In this dissertation, as shown in Figure 8.1, we provided a holistic approach to using tests as a useful source of information on requirements and we developed an end-to-end automated process to benefit from the testing phase during the development and maintenance of a system. By forming a convection cycle between requirements and tests, we proposed novel ways to aid software maintenance.

It is important to note that, although we propose an end-to-end process where each chapter discussed work that built on the work in other chapters as shown in Figure 1.11, the work in each chapter in this dissertation can be used stand-alone, given the proper inputs are obtained using suitable techniques (including the ones described in this dissertation). As an example, although the input to the technique described in Chapter 4 can be obtained using our technique described in Chapter 3, it can also be obtained using other techniques in the literature. This makes the work in each chapter stand on its own, and be useful for stakeholders even though they don't use our process end-to-end.

Finally, although the scope of the work in this dissertation is partially limited to agile processes to take advantage of the one-to-one mapping between requirements and features, it can be expanded to apply to plan-driven processes too. For this, the techniques discussed across this dissertation can be complemented with an additional step that lets stakeholders provide the mapping between requirements and features (albeit it is a one-to-many relationship). Overall, the techniques discussed can be extended to use this mapping to take in the proper inputs and provide the proper outputs for requirements and

features separately. The research on obtaining such a mapping between requirements and features, and extending the techniques in this dissertation to use such a mapping are left as benefits considered within the scope of future outlook provided by this dissertation.

8.2 Future Outlook

In each of the previous chapters, we discussed several short-term future work related to different aspects of the work in this dissertation. These work were concerned more about specific technical details and improvements to the respective work in each chapter. Below, we propose longer-term research directions that build on the work in this dissertation.

8.2.1 Mapping Requirements and Features

As discussed in Section 2.2.4, some of the work in this dissertation (Chapters 4, 6 and 6) described techniques that build on the assumption that requirements and features map one-to-one in a software system. As discussed in Section 2.2.4, this mapping may not be one-to-one for some systems, especially those that follow a plan driven process. The work in those chapters can be extended for plan driven processes too. The research in this direction needs to address the gap in mapping requirements and features concretely, e.g. through the use of an intermediate technique that links requirements and features. Once such a mapping is provided, the techniques described in this dissertation can use the links between requirements and features to develop similar outputs already provided here for agile processes.

A related improvement on extending the work in this dissertation is on the ability to group requirements/features. Many of the techniques in the literature and in this dissertation consider the requirements/features on the same level. However, the ability to group them is important in providing a top-down approach to software development and comprehension for stakeholders. Such grouping can be a step in addressing the mapping of requirements to features, since typically requirements and features have a many-to-many relationship, i.e. a requirement may be implemented via multiple features

and there may be multiple requirements implemented by a single feature. Grouping would allow addressing these relationships.

This is an important area of research, because there are many systems built using plan driven processes.

8.2.2 Addressing Non-Functional Requirements

Much work in the research literature and several chapters of this dissertation (Chapters 3, 4, and 6, through the use of features) focuses only on functional requirements. However, addressing the same concerns on non-functional requirements is a very important research area, since non-functional requirements play a critical role on the quality attributes of a software system. Some of the work in this dissertation can be improved to encompass non-functional requirements along with the functional ones by combining existing work and the techniques described here, and further improving the success of the techniques that work on non-functional requirements.

8.2.3 Considering the Valuation of Tests

Many of the chapters in this dissertation make use of tests as input (e.g. Chapter 7). The techniques provided currently assume that tests have equal importance with respect to what they are implemented to test. However, this may not hold for all systems, some tests may be considered more important because they demonstrate a critical behavior or property of the system. An important research direction is to integrate such different valuations of tests into the techniques in this dissertation. This can be achieved by using existing research in the field of “test suite minimization”, where a set of ‘important’ tests are selected to represent the whole collection of tests of a system based on some definition of being important.

8.2.4 Integration Into Production Development Environments

This dissertation provided a vision on using the techniques provided here as an end-to-end process as part of the software development lifecycle. All of the techniques proposed here output artifacts similar to the existing ones used in production systems nowadays, such as test coverage and automated documentation. Using the techniques provided here in real life production systems would not only demonstrate the adoption speed, but also pave the way for new research directions based on feedback and needs of the practitioners.

Bibliography

- [1] Apache Commons CLI. <http://commons.apache.org/cli>. Accessed: 30/05/2013.
- [2] Apache Log4J. <http://logging.apache.org/log4j/>. Accessed: 30/05/2013.
- [3] Apache Pool. <http://commons.apache.org/pool/>. Accessed: 30/05/2013.
- [4] ApacheLucene. <http://lucene.apache.org/java/docs/index.html>. Accessed: 30/05/2013.
- [5] AspectJ. <http://www.eclipse.org/aspectj/>. Accessed: 30/05/2013.
- [6] DOORS. <http://www-01.ibm.com/software/awdtools/doors/>. Accessed: 30/05/2013.
- [7] Eclipse AST Parser. <http://help.eclipse.org/helios/index.jsp?topic=/org.eclipse.jdt.doc.isv/reference/api/org/eclipse/jdt/core/dom/ASTParser.html>. Accessed: 30/05/2013.
- [8] FIT. <http://fit.c2.com/>. Accessed: 30/05/2013.
- [9] IBM Rational RequisitePro. <http://www-01.ibm.com/software/awdtools/reqpro/>. Accessed: 30/05/2013.
- [10] JUnit. <http://www.junit.org/>. Accessed: 30/05/2013.
- [11] MDA Guide Version 1.0.1. <http://www.omg.org/cgi-bin/doc?omg/03-06-01.pdf>. Accessed: 30/05/2013.
- [12] OMG: SysML Specification. <http://www.sysml.org/specs.htm>. Accessed: 30/05/2013.
- [13] Rakesh Agrawal, Tomasz Imieliski, and Arun Swami. Mining association rules between sets of items in large databases. *ACM SIGMOD Record*, 22(2):207–216, June 1993.

- [14] Rakesh Agrawal and Ramakrishnan Srikant. Fast Algorithms for Mining Association Rules in Large Databases. In J B Bocca, M Jarke, and C Zaniolo, editors, *Proceedings of the 20th International Conference on Very Large Data Bases*, volume 15 of *VLDB '94*, pages 487–499. Morgan Kaufmann Publishers Inc., Morgan Kaufmann Publishers Inc., 1994.
- [15] Neta Aizenbud-Reshef, Brian T. Nolan, Julia Rubin, and Yael Shaham-Gafni. Model Traceability. *IBM Systems Journal*, 45(3):515–526, 2006.
- [16] Ian Alexander. Towards Automatic Traceability in Industrial Practice. In *Proc of the 1st Int Workshop on Traceability*, pages 26–31. ScenarioPlus, UK, 2002.
- [17] Ian Alexander. Semi-Automatic Tracing of Requirement Versions to Use Cases: Experiences & Challenges. *Proceedings of the 2nd International Workshop on Traceability in Emerging Forms of Software Engineering TEFSE03*, 2003.
- [18] Giuliano Antoniol, Gerardo Canfora, Gerardo Casazza, Andrea De Lucia, and Ettore Merlo. Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering*, 28(10):970–983, October 2002.
- [19] Giuliano Antoniol and Yann Gael Gueheneuc. Feature Identification: An Epidemiological Metaphor. *IEEE Transactions on Software Engineering*, 32(9):627–641, September 2006.
- [20] Giuliano Antoniol, Jane Huffman Hayes, Yann-Gael Gueheneuc, and Massimiliano Di Penta. Reuse or rewrite: Combining textual, static, and dynamic analyses to assess the cost of keeping a system up-to-date. In Hong Mei and Kenny Wong, editors, *IEEE International Conference on Software Maintenance*, pages 147–156. IEEE, 2008.
- [21] Paul Arkley, Paul Mason, and Steve Riddle. Position paper: Enabling traceability. In *Proceedings of the 1st International Workshop on Traceability in Emerging Forms of Software Engineering*, pages 61–65, Edinburgh, Scotland, 2002.
- [22] Paul Arkley and Steve Riddle. Overcoming the traceability benefit problem, 2005.
- [23] Kent Beck. *Test-Driven Development By Example*, volume 2 of *The Addison-Wesley Signature Series*. Addison-Wesley, 2003.
- [24] Kent Beck and Martin Fowler. *Planning Extreme Programming*. Addison-Wesley, Boston, MA, USA, 1 edition, 2000.
- [25] Patrik Berander and Anneliese Andrews. Requirements Prioritization. In Aybüke Aurum and Claes Wohlin, editors, *Engineering and Managing Software Requirements*, number November, chapter 4, pages 69–94. Springer-Verlag, 2005.

- [26] Ted J. Biggerstaff, Bharat G. Mitbander, and Dallas Webster. The concept assignment problem in program understanding. In *Proceedings Working Conference on Reverse Engineering*, pages 27–43. IEEE Comput. Soc. Press, 1993.
- [27] David Binkley and Mark Harman. Locating dependence clusters and dependence pollution. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 177–186. IEEE, 2005.
- [28] Garret Birkhoff. *Lattice Theory*, volume 25 of *Colloquium Publications*. American Mathematical Society, 1967.
- [29] Barry Boehm and Richard Turner. *Balancing Agility and Discipline: A Guide for the Perplexed*, volume 22. Addison-Wesley, 2003.
- [30] Barry W. Boehm. *Software Engineering Economics*, volume SE-10 of *Prentice-Hall Advances in Computing Science & Technology Series*. Prentice Hall, 1981.
- [31] Barry W. Boehm. A spiral model of software development and enhancement. *Computer*, 21(5):61–72, 1988.
- [32] Scott Bradner. IETF RFC 2119: Key words for use in RFCs to Indicate Requirement Levels. 1999. *Internet Engineering Task Force*, 1997.
- [33] Hongyu Pei Breivold, Ivica Crnkovic, Rikard Land, and Stig Larsson. Using dependency model to support software architecture evolution. In *23rd IEEE/ACM International Conference on Automated Software Engineering - Workshops*, pages 82–91. IEEE, September 2008.
- [34] Sjaak Brinkkemper. Requirements engineering research the industry is and is not waiting for. In *Proceedings of the 10th International Workshop on Requirements Engineering: Foundation for Software Quality*, pages 41–54, 2004.
- [35] Frederick P. Brooks. No Silver Bullet - Essence and Accidents of Software Engineering. *IEEE Computer*, 20(4):10–19, 1987.
- [36] Trosky B Callo Arias, Pieter Spek, and Paris Avgeriou. A practice-driven systematic review of dependency analysis solutions. *Empirical Software Engineering*, 16(5):544–586, 2011.
- [37] Cyrus D. Cantrell. *Modern Mathematical Methods for Physicists and Engineers*. Cambridge University Press, 2000.
- [38] Jaelson Castro, Manuel Kolp, and John Mylopoulos. Towards requirements-driven information systems engineering: the Tropos project. *Information Systems Journal*, 27(6):365–389, 2002.

- [39] Kunrong Chen and Vaclav Rajlich. Case study of feature location using dependence graph. In *Proceedings of the 8th International Workshop on Program Comprehension*, pages 241–247. IEEE Comput. Soc, 2000.
- [40] Dai Clegg and Richard Barker. *CASE method fast-track : a RAD approach*. Addison-Wesley, Reading, 1 edition, August 1994.
- [41] Standards Committee. IEEE Recommended Practice for Software Requirements Specifications, 1998.
- [42] John Cooke and Roger Stone. A formal development framework and its use to manage software production. *IEEE Colloquium on Tools and Techniques for Maintaining Traceability During Design*, page 10/1, December 1991.
- [43] Thomas A. Corbi. Program understanding: Challenge for the 1990s. *IBM Systems Journal*, 28(2):294–306, 1989.
- [44] Vittorio Cortellessa, Ivica Crnkovic, Fabrizio Marinelli, and Pasqualina Potena. Driving the selection of cots components on the basis of system requirements. *Proceedings of the twentysecond IEEEACM international conference on Automated software engineering ASE 07*, page 413, 2007.
- [45] Krzysztof Czarnecki and Simon Helsen. Classification of Model Transformation Approaches. *Architecture*, 45(3):1–17, 2003.
- [46] Krzysztof Czarnecki and Simon Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–645, 2006.
- [47] Asa G. Dahlstedt and Anne Persson. Requirements Interdependencies - Moulding the State of Research into a Research Agenda. *9th International Workshop on Requirements Engineering Foundation for Software Quality REFSQ03*, pages 55–64, 2003.
- [48] Anne Dardenne, Axel Van Lamsweerde, and Stephen Fickas. Goal-directed requirements acquisition. *Science of Computer Programming*, 20(1-2):3–50, 1993.
- [49] Alan M. Davis. *Software Requirements: Analysis and Specification*. Prentice Hall, Upper Saddle River, NJ, 1989.
- [50] Alan M. Davis. *Software Requirements: Objects, Functions and States*. Prentice Hall, 2 edition, 1993.
- [51] Scott Deerwester, Susan T Dumais, George W Furnas, Thomas K Landauer, and Richard Harshman. Indexing by latent semantic analysis. *Journal of the American Society for Information Science*, 41(6):391–407, 1990.

- [52] Philippe Desfray. MDA When a major software industry trend meets our toolset , implemented since 1994 . Benefits of the MDA approach. *October*, pages 1–13, 2001.
- [53] Michel Marie Deza and Elena Deza. *Encyclopedia of Distances*, volume 2006. Springer Berlin Heidelberg, 2009.
- [54] Jeremy Dick. Rich Traceability. In *Proceedings of the 1st International Workshop on Traceability in Emerging Forms of Software Engineering*, pages 18–23, Edinburgh, Scotland, 2002. ACM.
- [55] Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. Feature location in source code: a taxonomy and survey. *Journal of Software: Evolution and Process*, 25(1):53–95, 2013.
- [56] Wolfgang Droschel and Manuela Wiemers. Das V-Modell 97, Der Standard für die Entwicklung von IT-Systemen mit Anleitung für den Praxiseinsatz. Technical report, Oldenbourg, 2000.
- [57] Eric Dubois, Jacques Hagelstein, and Andre Rifaut. Formal requirements engineering with ERAE. *Philips Journal of Research*, 43.4, 1989.
- [58] Christof Ebert and Reiner Dumke. *Software Measurement: Establish - Extract - Evaluate - Execute*. Springer-Verlag New York Inc., Secaucus, NJ, USA, August 2007.
- [59] Dennis Edwards, Sharon Simmons, and Norman Wilde. An approach to feature location in distributed systems. *Journal of Systems and Software*, 79(1):57–68, 2006.
- [60] Alexander Egyed and Paul Grünbacher. Supporting software understanding with automated requirements traceability. *International Journal of Software Engineering*, 15:783, 2005.
- [61] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. Locating features in source code. *IEEE Transactions on Software Engineering*, 29(3):210–224, March 2003.
- [62] Andrew David Eisenberg and Kris De Volder. Dynamic feature traces: finding features in unfamiliar code. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 337–346. IEEE, 2005.
- [63] Stuart R. Faulk. Software Requirements: A Tutorial. *Software Requirements Engineering 2nd Edition*, 1995.
- [64] Richard K. Fjeldstad and William T. Hamlen. Application program maintenance study: Report to our respondents. *Proceedings Guide*, 48, 1983.

- [65] Ismenia Galvao and Arda Goknil. Survey of Traceability Approaches in Model-Driven Engineering. In *11th IEEE International Enterprise Distributed Object Computing Conference EDOC 2007*, volume 07, pages 313–313. IEEE, 2007.
- [66] Bernhard Ganter and Rudolf Wille. *Formal concept analysis: mathematical foundations*, volume 1. Springer, Berlin and New York, 1999.
- [67] Martin Glinz. On Non-Functional Requirements. In *15th IEEE International Requirements Engineering Conference (RE 2007)*, pages 21–26. IEEE, October 2007.
- [68] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, The (3rd Edition)*. Java Series. Addison Wesley, 2005.
- [69] Orlena C. Z. Gotel and Anthony Finkelstein. An analysis of the requirements traceability problem. *Proceedings of IEEE International Conference on Requirements Engineering*, pages 94–101, 1994.
- [70] Sol J. Greenspan, Alexander Borgida, and John Mylopoulos. A Requirements Modeling Language and its Logic. *Information Systems Journal*, 11(1):9–23, 1986.
- [71] Orla Greevy, Stephane Ducasse, and Tudor Girba. Analyzing feature traces to incorporate the semantics of change in software evolution analysis. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 347–356. IEEE, 2005.
- [72] M. Hammond, C. Walker, and M. Moeller. Attacking the Quality Monster. *PC WEEK*, page 18, December 1998.
- [73] Abdelwahab Hamou-Lhadj and Timothy Lethbridge. Summarizing the Content of Large Traces to Facilitate the Understanding of the Behaviour of a Software System. *14th IEEE International Conference on Program Comprehension ICPC06*, pages 181–190, 2006.
- [74] Jane Huffman Hayes, Alex Dekhtyar, and David S. Janzen. Towards traceable test-driven development. In *ICSE Workshop on Traceability in Emerging Forms of Software Engineering*, pages 26–30. IEEE, May 2009.
- [75] Jane Huffman Hayes, Alex Dekhtyar, and Senthil Karthikeyan Sundaram. Advancing candidate link generation for requirements tracing: the study of methods, 2006.
- [76] David Helperin and Bill Hetzel. Software Quality Engineering. In *Fourth International Conference on Software Testing*, Washington DC, USA, 1987.

- [77] Institute of Electrical and Electronics Engineers. IEEE Standard Glossary of Software Engineering Terminology. 121990(1):1, 1990.
- [78] Justin Jackson. A Keyphrase Based Traceability Scheme. In *Tools and Techniques for Maintaining Traceability During Design IEE Colloquium on*, page 2. IET, 1991.
- [79] Ivar Jacobson, Grady Booch, and James Rumbaugh. *The Unified Software Development Process*. Addison Wesley, Reading, 1999.
- [80] Judit Jasz, Arpad Beszedes, Tibor Gyimothy, and Vaclav Rajlich. Static Execute After/Before as a replacement of traditional software dependencies. In *IEEE International Conference on Software Maintenance*, pages 137–146. IEEE, September 2008.
- [81] Karen Spärck Jones. A statistical interpretation of term specificity and its application in retrieval. *Journal of Documentation*, 60(5):493–502, 2004.
- [82] Ivan J. Jureta, Alex Borgida, Neil A. Ernst, and John Mylopoulos. *Techne: Towards a New Generation of Requirements Modeling Languages with Goals, Preferences, and Inconsistency Handling*, 2010.
- [83] Stephen Kan. *Metrics and Models in Software Quality Engineering*. {Addison-Wesley Professional}, 2 edition, 2002.
- [84] Noriaki Kano, Nobuhiko Seraku, Fumio Takahashi, and Shinichi Tsuji. Attractive quality and must-be quality. *Journal of the Japanese Society for Quality Control*, 14(2):39–48, 1984.
- [85] Jay Kothari, Trip Denton, Ali Shokoufandeh, and Spiros Mancoridis. Reducing Program Comprehension Effort in Evolving Software by Recognizing Feature Implementation Convergence. *15th IEEE International Conference on Program Comprehension ICPC 07*, pages 17–26, 2007.
- [86] Gerald Kotonya and Ian Sommerville. *Requirements Engineering: Processes and Techniques*. John Wiley & Sons, 1998.
- [87] Solomon Kullback and Richard A. Leibler. On information and sufficiency. *The Annals of Mathematical Statistics*, 22(1):79–86, 1951.
- [88] Patricia Lago, Henry Muccini, and Hans Van Vliet. A scoped approach to traceability management. *Journal of Systems and Software*, 82(1):168–182, 2009.
- [89] Craig Larman and Victor R. Basili. A History of Iterative and Incremental Development. *IEEE Computer*, pages 47–56, June 2003.

- [90] Greg Law. Cambridge University Study States Software Bugs cost Economy \$312 Billion per Year. Technical report, Cambridge University and Undo Software.
- [91] James Law and Gregg Rothermel. Incremental dynamic impact analysis for evolving software systems. In *14th International Symposium on Software Reliability Engineering, 2003. ISSRE 2003.*, pages 430–441. IEEE, 2003.
- [92] F. William Lawvere. Metric Spaces, Generalised Logic, and Closed Categories. *Milan Journal of Mathematics*, 43(1):136–166, 1973.
- [93] Dean Leffingwell and Don Widrig. *Managing software requirements: a unified approach*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, June 1999.
- [94] Do Prado Leite, Julio Cesar Sampaio, and Jorge H. Doorn. *Perspectives on Software Requirements*. Kluwer Academic Publishers, 2004.
- [95] Adrian Lienhard, Orla Greevy, and Oscar Nierstrasz. Tracking Objects to Detect Feature Dependencies. In *15th IEEE International Conference on Program Comprehension (ICPC '07)*, pages 59–68. IEEE, 2007.
- [96] Bennet P. Lientz, E. Burton Swanson, and Gail E. Tompkins. Characteristics of application software maintenance. *Communications of the ACM*, 21(6):466–471, June 1978.
- [97] Dapeng Liu, Andrian Marcus, Denys Poshyvanyk, and Vaclav Rajlich. Feature location via information retrieval based filtering of a single scenario execution trace. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering - ASE '07*, page 234, New York, New York, USA, November 2007. ACM Press.
- [98] Marco Lormans and Arie Van Deursen. Can LSI help Reconstructing Requirements Traceability in Design and Test? In *Proceedings of the 10th European Conference on Software Maintenance and Reengineering CSMR'06*, pages 47–56. IEEE Computer Society, 2006.
- [99] Marco Lormans, Hylke Van Dijk, Arie Van Deursen, Eric Nocker, and Aart de Zeeuw. Managing evolving requirements in an outsourcing context: an industrial experience report. In *Proceedings of the 7th International Workshop on Principles of Software Evolution*, pages 149–158. IEEE, 2004.
- [100] Andrea De Lucia, Fausto Fasano, Rocco Oliveto, and Genoveffa Tortora. Recovering traceability links in software artifact management systems using information retrieval methods. *ACM Transactions on Software Engineering and Methodology*, 16(4):13–es, September 2007.

- [101] Patrick Mader and Alexander Egyed. Do Software Engineers Benefit from Source Code Navigation with Traceability: An Experiment in Software Change Management. In *Proceedings of Automated Software Engineering*, pages 444–447, 2011.
- [102] Patrick Mader and Alexander Egyed. Assessing the effect of requirements traceability for software maintenance. In *28th IEEE International Conference on Software Maintenance*, pages 171–180. IEEE, 2012.
- [103] Prasanta Chandra Mahalanobis. On the generalised distance in statistics. In *Proceedings of the National Institute of Sciences of India*, pages 49–55, 1936.
- [104] Andrian Marcus and Jonathan I. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *25th International Conference on Software Engineering 2003 Proceedings*, volume 6, pages 125–135. Ieee, 2003.
- [105] Andrian Marcus, Jonathan I. Maletic, and Andrey Sergeyev. Recovery of traceability links between software documentation and source code. *International Journal of Software Engineering and Knowledge Engineering*, 15(5):811–836, 2005.
- [106] Andrian Marcus, Vaclav Rajlich, Joseph Buchta, Maksym Petrenko, and Andrey Sergeyev. Static Techniques for Concept Location in Object-Oriented Code. In *Proceedings of the 13th International Workshop on Program Comprehension*, pages 33–42. IEEE, IEEE Computer Society, 2005.
- [107] E Michael Maximilien and Laurie Williams. Assessing test-driven development at IBM. In *Proceedings of the 25th International Conference on Software Engineering*, volume 6 of *ICSE '03*, pages 564–569. IEEE Computer Society Washington, DC, USA, IEEE Computer Society, 2003.
- [108] Steve McConnell. *Rapid Development: Taming Wild Software Schedules*, volume 6. Microsoft Press, 1996.
- [109] Collin McMillan, Denys Poshyvanyk, and Meghan Revelle. Combining textual and structural analysis of software artifacts for traceability link recovery, 2009.
- [110] Joan Miller and Clifford Maloney. Systematic mistake analysis of digital computer programs. *Communications of the ACM*, 6(2):58–63, 1963.
- [111] Glenford J. Myers, Corey Sandler, and Tom Badgett. *The Art of Software Testing*. Wiley Publishing, 2004.
- [112] John Mylopoulos, Alex Borgida, Matthias Jarke, and Manolis Koubarakis. Telos: representing knowledge about information systems. *ACM Transactions on Information Systems*, 8(4):325–362, October 1990.

- [113] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 1999.
- [114] Dan North. Behavior Driven Development. <http://dannorth.net/introducing-bdd/>. Accessed: 30/05/2013.
- [115] Qvt Omg. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, 2008.
- [116] Floyd Guyten Patterson Jr. System Engineering Life Cycles: Life Cycles for Research, Development, Test, and Evaluation; Acquisition; and Planning and Marketing. In A. Sage and W. Rouse, editors, *Handbook of Systems Engineering and Management*, chapter 1, pages 59–111. Wiley, New York, New York, USA, 1999.
- [117] John L. Pfaltz. Using Concept Lattices to Uncover Causal Dependencies in Software. *4th International Conference Formal Concept Analysis*, 3874:233–247, 2006.
- [118] Francisco A. C. Pinheiro and Joseph A. Goguen. An object-oriented tool for tracing requirements, 1996.
- [119] Denys Poshyvanyk, Y.-G. Gueheneuc, Adrian Marcus, Giuliano Antoniol, and Vaclav Rajlich. Feature Location Using Probabilistic Ranking of Methods Based on Execution Scenarios and Information Retrieval. *IEEE Transactions on Software Engineering*, 33(6):420–432, June 2007.
- [120] V Rajlich and N Wilde. The role of concepts in program comprehension. In *Proceedings of the 10th International Workshop on Program Comprehension*, volume 0, pages 271–278. IEEE, IEEE, 2002.
- [121] Balasubramaniam Ramesh and Michael Edwards. Issues in the development of a requirements traceability model, 1993.
- [122] Matthias Riebisch. Towards a More Precise Definition of Feature Models. *Modelling Variability for Object Oriented Product Lines*, 22(3):64–76, 2003.
- [123] Martin P. Robillard. Automatic generation of suggestions for program investigation. *ACM SIGSOFT Software Engineering Notes*, 30(5):11, September 2005.
- [124] Martin P. Robillard. Topology analysis of software dependencies. *ACM Transactions on Software Engineering and Methodology*, 17(4):1–36, August 2008.
- [125] Martin P. Robillard and Gail C. Murphy. Concern graphs. In *Proceedings of the 24th International Conference on Software Engineering - ICSE '02*, page 406, New York, New York, USA, May 2002. ACM Press.

- [126] Gruia-Catalin Roman. A taxonomy of current issues in requirements engineering. *Computer*, 18(4):14–23, April 1985.
- [127] Walker W. Royce. Managing the development of large software systems: concepts and techniques. In *WESCON*, pages 1–9. IEEE, 1970.
- [128] RtcA Inc. RTCA/DO-178B Software Considerations in Airborne Systems and Equipment Certification. *Washington DC RTCA Inc*, 1992.
- [129] Thomas L. Saaty. *Fundamentals of decision making and prority theory with the analytic hierarchy process*. RWS Publications, Pittsburgh, PA, 1 edition, 1994.
- [130] Maher Salah and Spiros Mancoridis. A hierarchy of dynamic software views: from object-interactions to feature-interactions. In *20th IEEE International Conference on Software Maintenance*, pages 72–81. IEEE, 2004.
- [131] Neeraj Sangal, Ev Jordan, Vineet Sinha, and Daniel Jackson. Using dependency models to manage complex software architecture. *ACM SIGPLAN Notices*, 40(10):167, October 2005.
- [132] Robert C. Seacord, Daniel Plakosh, and Grace A. Lewis. *Modernizing Legacy Systems: Software Technologies, Engineering Processes, and Business Practices*. Addison-Wesley Professional, 2003.
- [133] Ian Sommerville. *Software engineering (7th edition)*. Addison Wesley, 7th edition, 2004.
- [134] Ian Sommerville and Pete Sawyer. *Requirements Engineering: A Good Practice Guide*. John Wiley & Sons, 1997.
- [135] M Staats, W Deng, A Rajan, M P E Heimdahl, and K Woodham. ReqsCov: A Tool for Measuring Test-Adequacy over Requirements, 2008.
- [136] Judith A Stafford and Alexander L Wolf. Architecture-Level Dependence Analysis for Software Systems. *International Journal of Software Engineering and Knowledge Engineering*, 11(04):431–451, 2001.
- [137] Tetsuo Tamai and Mayumi Itakura Kamata. *Impact of Requirements Quality on Project Success or Failure*, volume 14 of *Lecture Notes in Business Information Processing*. Springer Berlin Heidelberg, 2009.
- [138] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. Introduction to Data Mining. *Journal of School Psychology*, 19(1):51–56, 2005.
- [139] Gregory Tassey. The Economic Impacts of Inadequate Infrastructure for Software Testing. Technical report, National Institute of Standards and Technology, 2002.

- [140] Tom Tourwé, Wim Codenie, Nick Boucart, and Vladimir Blagojević. Demystifying Release Definition: From Requirements Prioritization to Collaborative Value Quantification. *Lecture Notes in Software Engineering*, 5512:37–44, 2009.
- [141] Cornelis J. Van Rijsbergen. *Information Retrieval*, volume 30 of *The Kluwer International Series on information retrieval*. Butterworths, London, 1979.
- [142] M D A Guide Version, Alan Kennedy, Kennedy Carter, and William Frank X-change Technologies. MDA Guide Version 1.0.1. *Object Management Group*, 234(June):51, 2003.
- [143] Marlon Vieira and Debra Richardson. Analyzing dependencies in large component-based systems. In *Proceedings of the 17th International Conference on Automated Software Engineering*, pages 241–244. IEEE, 2002.
- [144] Robert Watkins and Mark Neal. Why and how of requirements tracing. *IEEE Software*, 11(4):104–106, July 1994.
- [145] Martin West. Quality function deployment in software development. *IEEE Colloquium on Tools and Techniques for Maintaining Traceability During Design*, pages 5/1–5/7, December 1991.
- [146] Elaine J. Weyuker. Axiomatizing software test data adequacy. *IEEE Transactions on Software Engineering*, SE-12(12):1128–1138, December 1986.
- [147] Karl E. Wiegers. *Software Requirements, Second Edition*. Microsoft Press, second edition, 2003.
- [148] Norman Wilde and Michael C. Scully. Software reconnaissance: Mapping program features to code. *Journal Of Software Maintenance Research And Practice*, 7(1):49–62, 1995.
- [149] Laurie Williams, E Michael Maximilien, and Mladen Vouk. Test-driven development as a defect-reduction practice. In *14th International Symposium on Software Reliability Engineering 2003 (ISSRE 2003)*, volume 0 of *ISSRE '03*, pages 34–45. IEEE Computer Society, IEEE Computer Society, 2003.
- [150] Stefan Winkler and Jens Pilgrim. A survey of traceability in requirements engineering and model-driven development. *Software Systems Modeling*, 9(4):529–565, 2009.
- [151] W. Eric Wong, Swapna S. Gokhale, and Joseph R. Horgan. Quantifying the closeness between program components and features. *Journal of Systems and Software*, 54(2):87–98, 2000.

- [152] W. Eric Wong, Swapna S. Gokhale, Joseph R. Horgan, and Kishor S. Trivedi. Locating program features using execution slices. In *Proceedings of the IEEE Symposium on Application-Specific Systems and Software Engineering and Technology (ASSET'99)*, pages 194–203. IEEE Comput. Soc, 1999.
- [153] Bin Xin and Xiangyu Zhang. Efficient online detection of dynamic control dependence. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'07)*, page 185, New York, New York, USA, July 2007. ACM Press.
- [154] Eric SK Yu. Towards Modeling and Reasoning Support for Early-Phase Requirements Engineering. In A C M Press, editor, *RE 97 Proceedings of the 3rd IEEE International Symposium on Requirements Engineering*, RE '97, pages 226–235. IEEE Computer Society, 1997.
- [155] Wei Zhao, Lu Zhang, Yin Liu, Jiasu Sun, and Fuqing Yang. SNIAFL: towards a static non-interactive approach to feature location. *Proceedings 26th International Conference on Software Engineering*, 15(2):195–226, 2006.
- [156] Xuchang Zou, Raffaella Settini, and Jane Cleland-Huang. Improving automated requirements trace retrieval: a study of term-based enhancement methods. *Empirical Software Engineering*, 15(2):119–146, 2009.