UNIVERSITÉ DE MONTRÉAL

ANALYSING SOURCE CODE STRUCTURE AND MINING SOFTWARE REPOSITORIES TO CREATE REQUIREMENTS TRACEABILITY LINKS

NASIR ALI
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

THÈSE PRÉSENTÉE EN VUE DE L'OBTENTION
DU DIPLÔME DE PHILOSOPHIÆ DOCTOR
(GÉNIE INFORMATIQUE)
DÉCEMBRE 2012

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Cette thèse intitulée :

ANALYSING SOURCE CODE STRUCTURE AND MINING SOFTWARE REPOSITORIES TO CREATE REQUIREMENTS TRACEABILITY LINKS

présentée par : ALI Nasir
en vue de l'obtention du diplôme de : Philosophiæ Doctor
a été dûment acceptée par le jury d'examen constitué de :

This dissertation is dedicated to my parents,
for their support in seeing this work through its completion. . . .

ACKNOWLEDGMENTS

First of all, I would like to thank almighty ALLAH. Without his wish nothing is possible. The completion of this dissertation was only possible with the contributions of many people. Foremost are the generous support, mentoring, and patience of my advisors.

I would like to take this opportunity to thank my main supervisor, Dr. Prof. Yann-Gaël Guéhéneuc for his encouragement, advices and inspiration throughout this research. I have learned much from him about the attitudes and skills for conducting research, collaborating with other colleagues, presenting ideas, and being open-minded. He brought forth this research and allowed me to extend my education beyond the formal studies leading to this dissertation.

I would like to thank my co-supervisor, Prof. Dr. Giuliano Antoniol, whose support and guidance made my thesis work possible. I am very obliged for his motivation and immense knowledge in Software Engineering that, taken together, make him a great mentor. He always helped me to meet the short deadline by helping at evenings and–or on weekends. He never charged more than $5 for million dollar ideas and support. He could not even realise how much I have learned from him.

I would also thank Dr. Bram Adams, assistant professor at the École Polytechnique de Montréal, Canada, Dr. Massimiliano Di Penta, associate professor at University of Sannio, Italy, of the University of Sannio, and Dr. Jane Huffman Hayes, Professor in Computer Science at the University of Kentucky, USA, for their assistance and feedback during the various phases of the dissertation.

I would also like to thank the members of my Ph.D. committee who enthusiastically accepted to monitor and read my dissertation.

I am very thankful to all my colleagues of SOCCERLab and Ptidej teams for their feedback and the productive discussions. A special acknowledgment and thanks is given the members of all the software engineering groups, especially Ptidej and SOCCERLab teams, at Department of computing and software engineering at the École Polytechnique de Montréal who participated in the experiments of my ideas. I am truly indebted to them for their extra work during their studies.

Last but not the least, I am deeply grateful for the support my parents and siblings provided during every phase of this dissertation. The time they gave of their own personal lives to come here from across the world was a big help in my effort to complete the dissertation. A special thanks to my 10 years old nephew Harris Yaseen who always prayed that I become as intelligent as he is.

RÉSUMÉ

La traçabilité est le seul moyen de s'assurer que le code source d'un système est conforme aux exigences et que toutes ces exigences et uniquement celles-ci ont été implantées par les développeurs. Lors de la maintenance et de l'évolution, les développeurs ajoutent, suppriment ou modifient des fonctionnalités (y compris les fautes) dans le code source. Les liens de traçabilité deviennent alors obsolètes car les développeurs n'ont pas ou ne peuvent pas consacrer les efforts nécessaires pour les mettre à jour. Pourtant, la récupération de liens de traçabilité est une tâche ardue et coûteuse pour les développeurs. Par conséquent, nous trouvons dans la littérature des méthodes, des techniques et des outils pour récupérer ces liens de traçabilité automatiquement ou semi-automatiquement. Parmi les techniques proposées, la littérature montre que les techniques de recherche d'information (RI) peuvent récupérer automatiquement des liens de traçabilité entre les exigences écrites sous forme textuelle et le code source. Toutefois, la précision et le rappel des techniques RI sont affectés par certains facteurs qui influencent les entrées du processus de traçabilité des exigences. En raison de la faible précision et du faible rappel de ces techniques, la confiance des développeurs en l'efficacité des techniques de récupération des liens de traçabilité est affectée négativement.

Dans cette thèse, notre proposition est que l'ajout de nouvelles sources d'information et l'intégration des connaissances des développeurs à l'aide d'un modèle de confiance pourrait atténuer l'impact de ces facteurs et améliorer la précision et le rappel des techniques RI. Notre hypothèse est que la précision et le rappel des techniques RI pourraient être améliorés si deux (ou plus) sources d'information confirment un lien de traçabilité. Nous utilisons les données des référentiels logiciels, les relations entre classes, le partitionnement du code source, et les connaissances des développeurs comme sources d'informations supplémentaires pour confirmer un lien. Nous proposons quatre nouvelles approches de détection des liens de traçabilité : Histrace, BCRTrace, Partrace et une dernière basée sur les connaissances des développeurs. Nous proposons un modèle de confiance, Trumo, inspiré des modèles de confiance Web, pour combiner les votes des experts. Nous proposons alors quatre approche de recouvrement des liens de traçabilité : Trustrace, LIBCROOS, COPARVO et une nouvelle méthode d'attribution de poids, qui utilisent les experts créés précédemment et Trumo. Les approches proposées utilisent une technique de recherche d'information pour créer des liens de référence et utilisent l'opinion des experts pour réévaluer ces liens de référence.

Nous montrons que l'utilisation de plusieurs sources d'information améliore la précision et le rappel des techniques RI pour la traçabilité des exigences. Les résultats obtenus dans cette thèse montrent une amélioration de jusqu'à 22% de précision, 7% de rappel et 83% de réduction d'effort des développeurs pour la suppression manuelle de liens faux positifs. Les résultats obtenus dans cette thèse sont prometteurs et nous espérons que d'autres recherches dans ce domaine pourraient améliorer notre précision et rappel encore plus.

ABSTRACT

Traceability is the only means to ensure that the source code of a system is consistent with its requirements and that all and only the specified requirements have been implemented. During software maintenance and evolution, as developers add, remove, or modify features (including bugs), requirement traceability links become obsolete because developers do not/cannot devote effort to update them. Yet, recovering these traceability links later is a daunting and costly task for developers. Consequently, the literature proposed methods, techniques, and tools to recover semi-automatically or automatically these traceability links. Among the proposed techniques, the literature showed that information retrieval (IR) techniques can automatically recover traceability links between free-text requirements and source code. However, precision and recall of IR techniques are impacted by some factors, which impact the input of requirements traceability process. Due to low precision and–or recall, developers' confidence in the effectiveness of traceability link recovery technique is negatively affected.

In this dissertation, our thesis is that adding more sources of information using a trust-based model and integrating developers' knowledge in automated IR-based requirements traceability approaches could mitigate the impact of the factors and improve the precision and recall of IR techniques. Our conjecture is that the accuracy of information retrieval techniques could be improved if two (or more) sources of information vote for a link. We use software repositories' data, binary class relationships, source code partitioning, and developer's knowledge as extra sources of information to confirm a link. We propose four approaches, Histrace, BCRTrace, Partrace, and developer's knowledge, to create experts out of the available extra sources of information. We propose a trust-model, Trumo, inspired by Web trust-models of users, to combine the experts' votes. We then propose four traceability link recovery approaches: Trustrace, LIBCROOS, COPARVO, and an improved term weighting scheme, which use the experts created by previous four techniques and Trumo. The proposed approaches use an IR technique to create the baseline links and use experts' opinions to reevaluate baseline links.

We show that using more sources of information improve the accuracy of IR techniques for requirements traceability. The achieved results in this dissertation show up to 22% precision, 7% recall improvement and 83% reduction in developer's effort for manually removing false-positive links. The results achieved in this dissertation are promising and we hope that further research in this field might improve the accuracy of IR techniques more.

TABLE OF CONTENTS

DEDICATION . . . iii

ACKNOWLEDGMENTS . . . iv

RÉSUMÉ . . . v

ABSTRACT . . . vii

TABLE OF CONTENTS . . . viii

LIST OF TABLES . . . xiii

LIST OF FIGURES . . . xiv

LIST OF APPENDICES . . . xv

LIST OF ABBREVATIONS . . . xvi

CHAPTER 1 INTRODUCTION . . . 1
- 1.1 Problem and Motivation . . . 3
- 1.2 Creation of Experts from Each Source of Information . . . 5
- 1.3 Combining Experts' Opinions . . . 6
- 1.4 Usage of Experts And Their Opinions . . . 6
- 1.5 Tool Support . . . 7
- 1.6 Organisation of the Dissertation . . . 9

I Background 12

CHAPTER 2 INFORMATION RETRIEVAL PROCESS . . . 13
- 2.1 IR Process . . . 13
  - 2.1.1 Pre-Processing . . . 14
  - 2.1.2 Existing Term Weighting . . . 14
- 2.2 IR Techniques . . . 15
  - 2.2.1 Vector Space Model . . . 16
  - 2.2.2 Latent Semantic Indexing . . . 16

2.2.3 Latent Dirichlet Allocation . . . 17
2.2.4 Jensen-Shannon Divergence Model . . . 18
2.3 Generation of Traceability Links' Sets . . . 19
2.4 IR Performance Measures . . . 20
2.4.1 Precision and Recall . . . 20
2.4.2 F-Measure . . . 20
2.5 Statistical Hypothesis Testing . . . 21
2.5.1 Statistical Tests . . . 22

CHAPTER 3 RELATED WORK . . . 23
3.1 Traceability Approaches . . . 23
3.2 Feature Location . . . 26
3.3 Binary Class Relationships . . . 27
3.4 Web Trust Model . . . 28
3.5 Eye-Tracking . . . 29
3.6 Summary of Related Approaches . . . 30

II Creation of Experts and Combining their Opinions 32

CHAPTER 4 CREATION OF EXPERTS . . . 33
4.1 Histrace . . . 33
4.2 BCRTrace . . . 35
4.3 Partrace . . . 37
4.4 Developer's Knowledge . . . 38
4.5 Summary . . . 39

CHAPTER 5 COMBINING EXPERTS' OPINIONS . . . 40
5.1 Definitions . . . 40
5.2 Trumo . . . 41
5.3 Voting . . . 42
5.4 Weighting Technique . . . 43
5.4.1 Static Weight . . . 43
5.4.2 DynWing . . . 44
5.4.3 Principal Component Analysis . . . 45
5.5 Combination of Experts, Trumo, and Weighting Techniques . . . 45

III Usage of Experts and their Opinions 46

CHAPTER 6 ASSESSING TRUSTRACE AS A TRACEABILITY RECOVERY METHOD 47
- 6.1 Trustrace: Trust-based Traceability . . . 48
  - 6.1.1 Usage of Histrace . . . 49
- 6.2 Empirical Evaluation . . . 49
  - 6.2.1 Goal . . . 50
  - 6.2.2 Research Questions . . . 50
  - 6.2.3 Pre-Processing . . . 51
  - 6.2.4 Building Sets of Traceability Links . . . 53
  - 6.2.5 Experimental Settings . . . 53
  - 6.2.6 Analysis Method . . . 54
- 6.3 Results . . . 55
- 6.4 Discussion . . . 58
  - 6.4.1 Dataset Quality Analysis . . . 58
  - 6.4.2 DynWing vs. MSW vs. PCA . . . 62
  - 6.4.3 Number of Experts . . . 63
  - 6.4.4 Other Observations . . . 64
  - 6.4.5 Practical Applicability of Trustrace . . . 65
  - 6.4.6 Revisiting the Conjectures . . . 65
  - 6.4.7 Threats to Validity . . . 66
- 6.5 Summary . . . 66

CHAPTER 7 ASSESSING THE USEFULNESS OF TRUMO FOR BUG LOCATION 68
- 7.1 LIBCROOS . . . 69
  - 7.1.1 LIBCROOS Abstract Model . . . 69
  - 7.1.2 IR Engine . . . 70
  - 7.1.3 Usage of BCRTrace . . . 71
  - 7.1.4 $Trumo_{Ranker}$ . . . 71
- 7.2 Empirical Study . . . 71
  - 7.2.1 Research Questions, Hypothesis, and Variables . . . 72
  - 7.2.2 Preparing Datasets for Experiments . . . 73
  - 7.2.3 Linking Bugs Reports and Classes using VSM . . . 74
  - 7.2.4 Linking Bugs Reports and Classes using LSI . . . 74
  - 7.2.5 Linking Bugs Reports and Classes using LIBCROOS . . . 74
  - 7.2.6 Analysis Methods . . . 75
- 7.3 Results and Discussion . . . 75

7.3.1 Threats to Validity 78
7.4 Summary 80

CHAPTER 8 IMPLEMENTATION OF COPARVO-BASED REQUIREMENTS TRACEABILITY 81
8.1 COPARVO 82
8.1.1 Usage of Partrace 82
8.2 Empirical Study 83
8.2.1 Goal 84
8.2.2 Research Questions 84
8.2.3 Variables and Objects 84
8.2.4 Usage of COPARVO 85
8.2.5 Analysis Method 86
8.3 Experiment Results 87
8.4 Discussion & Qualitative Analysis 90
8.4.1 Effort Reduction 90
8.4.2 Ascertaining Experts 90
8.4.3 Different Scenarios 90
8.4.4 Role of Identifiers 91
8.4.5 Threats to Validity 92
8.5 Summary 93

CHAPTER 9 USING DEVELOPERS' KNOWLEDGE FOR IMPROVING TERM WEIGHTING SCHEME 94
9.1 Empirical Study 95
9.2 Experiment Design: Eye-Tracking 96
9.2.1 Eye-tracking System 97
9.2.2 Experiment Settings 97
9.2.3 Subjects Selection 97
9.2.4 Source Code Selection 98
9.2.5 Links, Tasks, and Questionnaires 99
9.2.6 Procedure 99
9.2.7 Analysis And Result 100
9.3 Experiment Design: SE/IDF and DOI/IDF 103
9.3.1 Objects 104
9.3.2 Procedure 104
9.3.3 Analysis Method 106

9.3.4 Results 107
9.4 Discussion 109
9.4.1 Identifiers' Vocabulary 110
9.4.2 Male vs. Female 111
9.5 Threats to Validity 111
9.6 Summary 112

IV Conclusion and Future Work 114

CHAPTER 10 CONCLUSION 115
10.0.1 Creation of Experts 115
10.0.2 Combining Experts' Opinion 116
10.0.3 Usage of Experts and their Opinions 116
10.1 Limitations 118
10.2 Future Work 119

REFERENCES 121

APPENDICES 131

LIST OF TABLES

Table 1.1 Average precision and recall range of RTAs. The bold values show the extreme cases of precision and recall 2

Table 3.1 Related work summary of closely related approaches to the work presented in this dissertation. CTW represents customised term weighting, SCP represents source code partitioning, Mul. Exp. represents the option to add multiple experts, AEW represents automated expert weighting,ET, WTM supp., Fl, and RT represent eye-tracker, web trust model support, feature location and requirements traceability respectively 31

Table 4.1 Source Code Sections used in Experimentation 38

Table 5.1 Different combinations of experts, Trumo, and weighting techniques . . 45

Table 6.1 Precision and recall values for *jEdit*, *Pooka*, *Rhino*, and *SIP*, Mann-whitney test results, and Cliff's d results 57

Table 6.2 F_1 values for *jEdit*, *Pooka*, *Rhino*, and *SIP*, and Mann-whitney test results 61

Table 7.1 Descriptive statistics LIBCROOS, LSI, and VSM. LIBC. and SD represent LIBCROOS and Standard Deviation respectively 77

Table 8.1 COPARVO, VSM, and Other Source Code Partitions' Combination Results (Bold values represent the top experts (β_i) voting results). (+) sign represents the combination of source code partitions, whereas (-) sign represents different experts' voting on recovered links by VSM (COPARVO) 89

Table 8.2 COPARVO and VSM Total Recovered Links at 0 threshold and p-values of F-Measure 91

Table 9.1 Questions of the Eye-Tracking Experiment). 99

Table 9.2 Average time row shows average time spent on each SCE. Ranking row shows the ranking of each SCE based on its importance/fixations (4 means the most important). 102

Table 9.3 Average Precision, Recall, and F-measure values and Wilcoxon p-values.105

LIST OF FIGURES

Figure 1.1 Inputs of traceability approach and impacting factors 4
Figure 1.2 Excerpt of Factrace GUI 9
Figure 2.1 IR-based RT Links Recovery Process 13
Figure 5.1 Overlapping of $R2C$, $R2CT_{i,r_j,t_k}$, and Tr 42
Figure 6.1 Trust-based requirement traceability process 48
Figure 6.2 Excerpt of *Pooka* SVN Log 53
Figure 6.3 Precision and recall values of JSM, Trustrace, and VSM, with the threshold t varying from 0.01 to 1 by step of 0.01. The X axis shows recall and Y axis shows precision. 56
Figure 6.4 Precision and recall values, with the threshold t varying from 0.01 to 1 by steps of 0.01. The X axis shows precision values and Y axis recall values. DW represents the DynWing results. 59
Figure 6.5 Similarity between merged requirements and source code documents . . 60
Figure 6.6 Rhino precision and recall graph for $Trustrace$ using one and two experts. The X axis shows precision values and the Y axis recall values. . 63
Figure 7.1 High-level Diagram of LIBCROOS 70
Figure 7.2 LIBCROOS, LSI, and VSM Results Boxplot Graph 76
Figure 8.1 Top Experts for Traceability Link Recovery 87
Figure 8.2 F-measure values at different level of threshold. These graphs only shows the highest F-measure results in different categories for the sake of simplicity. 88
Figure 9.1 FaceLAB eye-tracker excerpt 98
Figure 9.2 Distribution of Average Correct Answers and Average Fixations 101
Figure 9.3 A heatmap showing the cumulative fixations of subjects. The colors red, orange, green and blue indicate the decrease in number of fixations from highest to lowest. 102
Figure 9.4 $LSI_{TF/IDF}$, $LSI_{SE/IDF}$, and $LSI_{DOI/IDF}$ F-measure values of iTrust and Pooka 108

LIST OF APPENDICES

Appendix A DATASETS STATISTICS 131
Appendix B LIST OF PUBLICATIONS 134

LIST OF ABBREVATIONS

AOI	Area of Interest
AST	Abstract Syntax Tree
BCR	Binary Class Relationship
BLT	Bug Location Techniques
BRCG	Branch Reserving Call Graph
Coparvo	Code Partitioning and Voting
DOI/IDF	Domain Or Implementation concepts/Inverse Document Frequency
DynWing	Dynamic Weighting
FacTrace	Artefacts Traceability Tool
Histrace	Software Histories Traces
IDF	Inverse Document Frequency
IR	Information Retrieval
JSM	Jensen-Shannon Divergence Model
LDA	Latent Dirichlet Allocation
LIBCROOS	LInguistic (textual) and BCRs of Object-Oriented Systems
LOC	Lines of Code
LSI	Latent Semantic Indexing
MSW	Multiple Static Weights
OO	Object-Oritented
PADL	Patterns and Abstract-level Description Language
Ptidej	Patterns Trace Identification, Detection, and Enhancement for Java
RT	Requirements Traceability
RTA	Requirements Traceability Approach
SCE	Source Code Entity
SCP	Source Code Partition
SE/IDF	Source Code Entity/Inverse Document Frequency
SVD	Singular Value Decomposition
TF	Term Frequency
Trumo	Trust Model
Trustrace	Trust-based Traceability
TTP	Trusted Third Party
VSM	Vector Space Model

CHAPTER 1

INTRODUCTION

Preliminary to any software evolution task, a developer must comprehend the project landscape (Dagenais *et al.* (2010)), in particular, the system architecture, design, implementation, and the relations between the various artifacts using any available documentation. Program comprehension occurs in a bottom-up manner (Pennington (1987)), a top-down manner (Brooks (1983)), or some combination thereof (Mayrhauser and Vans (1993)). Developers use different types of knowledge during program comprehension, ranging from domain-specific knowledge to general programming knowledge. Traceability links between source code and sections of the documentation, *e.g.*, requirements, aid both top-down and bottom-up comprehension (De Lucia *et al.* (2006)).

Requirement traceability is defined as "the ability to describe and follow the life of a requirement, in both a forwards and backwards direction (*i.e.*, from its origins, through its development and specification, to its subsequent deployment and use, and through all periods of on-going refinement and iteration in any of these phases)" (Gotel and Finkelstein (1994)). Traceability links between the requirements of a system and its source code are helpful in reducing comprehension effort. This traceability information also helps in software maintenance and evolution tasks. For example, once a developer has traceability links, she can easily trace what software artifacts must be modified to implement a new requirement. Traceability links are also essential to ensure that a source code is consistent with its requirements and that all and only the specified requirements have been implemented by developers (Koo *et al.* (2005); Hayes *et al.* (2007)).

Despite the importance of traceability links, during software maintenance and evolution, as developers add, remove, or modify features, requirement traceability links become obsolete because developers do not/cannot devote effort to update them. This insufficient traceability information is one of the main factors that contributes to project over-runs, failure, and difficult to maintain (Ali *et al.* (2011b); Dömges and Pohl (1998); Leffingwell (1997)). Insufficient traceability information results in the need for costly and laborious tasks of manual recovery and maintenance of traceability links. These manual tasks may be required frequently depending on how frequently software systems evolve or are maintained.

Consequently, the literature proposed methods, techniques, and tools to recover semi-automatically or automatically traceability links. Researchers used information retrieval (IR) techniques, *e.g.*, (Abadi *et al.* (2008); Antoniol *et al.* (2002b); Marcus and Maletic (2003a)),

Table 1.1 Average precision and recall range of RTAs. The bold values show the extreme cases of precision and recall

	VSM		**LSI**		**JS**		**LDA**	
Datasets	Precision	Recall	Precision	Recall	Precision	Recall	Precision	Recall
SCA Abadi *et al.* (2008)	20 – 43	51 – 76	14 – 26	41 – 78	23 – 41	57 – 78	–	–
CORBA Abadi *et al.* (2008)	50 – **80**	68 – 89	11 – 50	14 – 61	43 – 65	55 – 81	–	–
MODIS Sundaram *et al.* (2005)	**7.9**	75.6	4.2 – 6.3	63.4 – 92.6	–	–	–	–
CM-1 Sundaram *et al.* (2005)	**1.5**	97.7	**0.9**	98.6 – **98.8**	–	–	–	–
Easy Clinic Oliveto *et al.* (2010)	17 – 80	4 – 90	17 – 60	3 – 90	17 – 80	4 – 91	9 – **40**	20 – **60**
eTour Oliveto *et al.* (2010)	17 – 68	5 – 47	17 – 64	4 – 46	17 – 76	5 – 47	**2** – 16	**1** – 20

to recover traceability links between high-level documents, *e.g.*, requirements, manual pages, and design documents, and low-level documents, *e.g.*, source code and UML diagrams. IR techniques assume that all software artifacts are/can be put in some textual format. Then, they compute the textual similarity between each two software artifacts, *e.g.*, the source code of a class and a requirement. A high textual similarity means that the two artifacts probably share several concepts and that, therefore, they are likely linked to one another.

The effectiveness of IR techniques is measured using IR metrics: recall, precision, or some average of both, *e.g.*, F_1 measure (Antoniol *et al.* (2002b); Hayes *et al.* (2004, 2008)). For a given requirement, recall is the percentage of recovered links over the total number of pertinent, expected links, while precision is the percentage of correctly recovered links over the total number of recovered links. High recall could be achieved by linking each requirement to all source code entities, but precision would be close to zero. A high precision could be achieved by reporting only obvious links, but recall would be close to zero. Either extreme cases are undesirable because developers then would need to manually review numerous candidate links to remove false positive links and–or study the source code to recover missing links (Antoniol *et al.* (2002b)).

The tradeoff between precision and recall depends on the context in which IR technique is used. For example, search engines prefer precision over recall because a user usually only looks at the top 10 to 30 retrieved Web pages related to her query. A search engine user may not care if a Web page related to her query is missing. In contrast to search engines, a software engineer prefers recall over precision in requirements traceability (RT) because a high recall reduces error-prone manual search for missing traceability links. Empirical studies (Ali *et al.* (2012a)) on the precision and recall of automated traceability link recovery techniques have shown that when a high recall, *e.g.*, 90%, is achieved then precision is usually less than 40% and in some cases is even less than 10%. Such low precision values in automated traceability-link recovery are a problem for developers. For example, 10% precision means that a software engineer must look on average through ten candidate links to find a true positive link. The low precision is likely to negatively affect the software engineers' trust in the accuracy of the automated traceability recovery approaches and increase manual effort.

1.1 Problem and Motivation

Table 1.1 summarises the precision and recall values of some requirements traceability approaches (RTAs) described in the literature are mainly: Vector Space Model (VSM), Latent Semantic Indexing (LSI), Rule-based, Jensen-Shannon similarity model (JSM), and Latent Dirichlet Allocation (LDA). It shows that, depending on the datasets, precision values vary from 0.9% to 95.9% and recall values vary from 3% to 99.8%. For example, Sundaram *et al.* (2005) achieved 1.5% to 7.9% precision with VSM whereas Abadi *et al.* (2008) achieved 50% to 80% precision with the same technique. Both group of researchers used standard VSM to obtain their results. The low precision and recall values are not only due to the use of one IR technique or another; the differences are due to the quality of RTAs' inputs, *i.e.*, requirements, source code, and developers' programming knowledge (Ali *et al.* (2012a)).

In our previous study (Ali *et al.* (2012a)), we performed an incremental literature review (ILR) to understand the cause of the variations in precision and recall values for different RTAs. We observed that there exist some factors that impact RTAs' inputs. We documented seven factors (see Figure 1.1), *i.e.*, ambiguous requirement, vague requirement, conflicting requirement, granularity level, identifiers' quality, domain knowledge, and programming language knowledge. We also showed that controlling, avoiding, and–or mitigating the impact of these factors could yield better RTAs accuracy. For each factor, we documented some preventive measures and–or techniques to improve the quality of RTA inputs.

Regardless the importance of controlling/avoiding RTAs' input impacting factors, for some factors, *e.g.*, identifiers' quality, it would not be feasible to completely control/avoid their impact on RTAs' inputs . Because the preventive measures and–or techniques (Ali *et al.* (2012a)) to improve the quality of RTA inputs are not 100% automated. A developer must manually verify the quality of all these inputs at the end of a RT process. For example, Dit *et al.* (2011a) performed an empirical study on feature location using different splitting algorithms. Their results showed that manually splitting identifiers could provide better results. However, manually splitting identifiers would require a lot of developer's effort. The authors argued that adding execution trace information could yield better accuracy than manually splitting identifiers. Poshyvanyk *et al.* (2007) also showed that for feature location, combining execution traces' information with IR techniques improves the accuracy of IR techniques. Many other researchers (Poshyvanyk *et al.* (2007); Dit *et al.* (2011a); Gethers

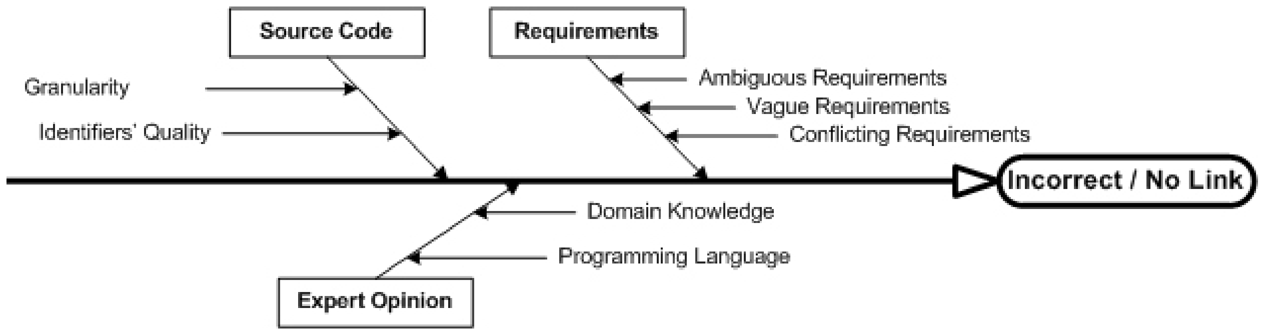


Figure 1.1 Inputs of traceability approach and impacting factors

et al. (2011)) also discussed the importance of adding more sources of information. However, their model to combine different experts was not generalised and they did not consider adding more than two unique sources of information. In addition, to the best of our knowledge, adding more sources of information in the field of requirements traceability has not been analysed yet. Thus, our thesis is:

Adding more sources of information and combining them with IR techniques could improve the accuracy, in terms of precision and recall, of IR techniques for requirements traceability.

To prove our thesis, we use each source of information as an expert to vote on some baseline links recovered by an IR technique. The higher the number of experts voting (Poshyvanyk *et al.* (2007); Ali *et al.* (2011b)) for a baseline link, the higher the confidence in the link. We get inspiration from Web models of users' trust (Berg and Van (2001); McKnight *et al.* (2002); Palmer *et al.* (2000); Koufaris and Hampton-Sosa (2004)): the more users buy from a Web merchant and–or refer the Web merchant, the higher the users' trust would be on the Web merchant. In other words, if more sources of information, *e.g.*, friends, family, online reviews, third party guarantees, refund policies etc., are "positively" recommending a Web merchant the higher an online buyer's trust would be. We apply the same idea to RTAs and use software repositories data, binary class relationship, source code entities, and developer's knowledge as extra source of information to trust a link recovered by an IR technique.

Following, we describe the steps to create experts using available sources of information. Each expert has its own trust-value for each baseline link. We propose a trust-model to combine the trust-values of each expert to reevaluate a baseline link. We propose different approaches to use the opinion of these experts.

1.2 Creation of Experts from Each Source of Information

We define four main approaches, *i.e.*, Histrace, LIBCROOS, COPARVO, and developer's knowledge, to create experts with the extra source of information.

Histrace: While developers may not evolve requirements in synchronisation with source code, they frequently update software repositories, *e.g.*, CVS/SVN, Bugzilla, mailing lists, forums, and blogs, to keep track and communicate about their changes. We conjecture that we can mine software repositories to build improved traceability link recovery techniques. We show that mining software repositories and combining the mined data with the results of some IR techniques allow recovering traceability links with better accuracy than when using IR techniques alone because of the multiple sources of information. To confirm our conjecture, we propose Histrace, a mining technique to create experts using CVS/SVN commits and bug reports and an IR-based technique. Histrace links, *e.g.*, $Histrace_{Bugs}$, act as expert to reevaluate baseline links recovered by IR technique.

BCRTrace: In some projects, software repositories are not available and–or not up-to-date. In this situation, Histrace may not be effective to create experts. Thus, we propose BCRTrace that uses Binary Class Relationships (BCRs) of Object-oriented (OO) systems, to create experts. Our conjecture behind BCRTrace is that when developers implement a feature, they usually use some BCRs among the classes playing role to implement a feature. BCRTrace uses BCR among the classes acts like an expert to reevaluate the baseline links.

Partrace: It is quite possible that developers may not use all BCRs to implement a feature. Thus, in this case BCRTrace may not be effective in creating experts. To handle such situation, we propose Partrace (Partitioning Source Code for Traceability). Our conjecture behind Partrace is that information extracted from different source code partition (*e.g.*, class names, comments, class variables, or methods signatures) are different information sources; they may have different level of reliability in RT and each information source may act as a different expert recommending traceability links. Thus, using each partition as a separate expert to vote on baseline links could yield better accuracy. Partrace divides a source code file into different partitions and each partition acts as an independent expert. In this dissertation, we divide source code into four parts, class name, method name, variable name, and comments. A developer could define the granularity of an expert. For example, if she wants to use each source code line as a separate partition/expert.

Developers' Knowledge: IR-based RTAs are automatic to recover traceability links. However, it does not free a developer to manually verify a link or recovery any missing link. It shows that developers' knowledge is still required and better than the automated techniques.

Thus, the knowledge of a developer could be used as extra source of information. We conjecture that understanding how developers verify RT links could help improve the accuracy of IR-based approaches to recover RT links. To confirm our conjecture, we use developers' knowledge as an extra source of information to integrate into automated techniques. We use an eye-tracking system to capture developers' eye movements while they verify traceability links. We observe which source code part has more importance than the other for a developer. We use an eye-tracker to capture developers' eye movement while they perform RT task. We integrate their source code preferences into IR weighting scheme.

1.3 Combining Experts' Opinions

We must combine each expert's opinion to reevaluate the similarity of baseline links recovered by an IR technique. To combine different experts' opinion, we propose a trust model, *i.e.*, Trumo. Trumo takes two inputs, *i.e.*, baseline RT links recovered by an IR technique and RT links recovered by experts, to generate a new set of trustable links. First, Trumo takes a baseline link created by an IR technique as an initial trustable link. Second, RT links recovered by experts, *e.g.*, CVS/SVN commits, vote on the baseline links. Each expert has its own trust-value for each baseline link. Trumo discards a link if no expert votes for it. Third, Trumo combines the opinions (trust-values) of all experts to generate a new trustable set of RT links. If more experts vote for a link then Trumo mark that link as more trustable than the others. Lastly, it uses a weighting scheme to assign weight to each expert and combine their opinions.

1.4 Usage of Experts And Their Opinions

Now that we have experts and trust-model to combine their opinions, we describe the usage of the experts and their opinions. Each approach is independent and use one expert, *e.g.*, Histrace or Partrace, at a time.

– We propose Trustrace, a trust-based traceability recovery approach, in terms of experts and trust-model. Trustrace uses software repositories' data as more sources of information to reevaluate a baseline link created by an IR technique. Trustrace uses experts created by Histrace and combine their opinion using Trumo. DynWing is a weighting technique to dynamically assign weights to each expert giving their opinions about the recovered links. The results of this contribution were published in Transactions in Software Engineering (Ali *et al.* (2012b)) and 19^{th} IEEE International conference on Program Comprehension (ICPC'11) (Ali *et al.* (2011b)).

– LIBCROOS uses BCRTrace to reevaluate baseline traceability links recovered by an IR technique. To exploit the benefits and application of Trumo, we customise it to integrate BCRTrace for bug location. Trumo provides a model to combine various experts, *e.g.*, software repositories. LIBCROOS helps to put culprit classes at the top in the ranked list. More BCRTrace expert vote for a link more Trumo puts that link at the top in the ranked list. The results of this contribution are published in 12th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'12) (Ali *et al.* (2012c)).

– COPARVO uses experts created by Partrace combined with an IR technique to create links between requirements and source code. Each expert created by Partrace, *e.g.*, class name or method name, could vote on the links recovered by an IR technique to remove false positive links. In COPARVO, at least two experts must agree on a link to keep the link. COPARVO uses a technique to identify top two experts that must agree on a baseline link. The results of this contribution were published in 18th Working Conference on Reverse Engineering (WCRE'11) (Ali *et al.* (2011a)).

– We use developer's knowledge observed during eye-tracking experiment as an extra source. We use this extra source of information to propose two new weighting schemes called SE/IDF (source code entity/inverse document frequency) and DOI/IDF (domain or implementation/inverse document frequency) to recover RT links combined with an IR technique. SE/IDF is based on the developers preferred source code entities (SCEs) to verify RT links. DOI/IDF is an extension of SE/IDF distinguishing domain and implementation concepts. We use LSI combined with SE/IDF, DOI/IDF, and TF/IDF to show, using two systems, iTrust and Pooka, that $LSI_{DOI/IDF}$ statistically improves the accuracy of the recovered RT links over $LSI_{TF/IDF}$. The results of this contribution were published in 28th IEEE International Conference on Software Maintenance (ICSM'12) (Ali *et al.* (2012d)).

1.5 Tool Support

The methods and techniques presented in this dissertation have been partially and–or fully implemented and integrated in FacTrace[1], for artifact TRACEability (Ali *et al.* (2010, 2011a,b, 2012c,d)). Figure 1.2 shows an excerpt of Factrace user interface. FacTrace is an

1. http://www.factrace.net

IR-based RT link recovery tool. It supports several IR techniques, namely VSM, LSI, LDA, and JSM. FacTrace allows developers to select the source and target artifacts to recover traceability links between them. FacTrace uses an external Java parser to extract all the source code identifiers for further processing. A developer can set various parameters in FacTrace to achieve desired results. For example, she can select the weighting schemes for an IR technique, IR technique, and thresholds etc. Threshold selection helps to retrieve only a set of traceability links whose similarity is above than a certain level.

FacTrace provides several modules that help from traceability recovery to traceability links verification. FacTrace aids software engineers in different tasks, namely, requirement elicitation, requirement analysis, artifact traceability, and trust-based traceability (Ali *et al.* (2011b)). FacTrace has a graphical interface to perform different tasks. Following sections give you a brief description of different features of FacTrace.

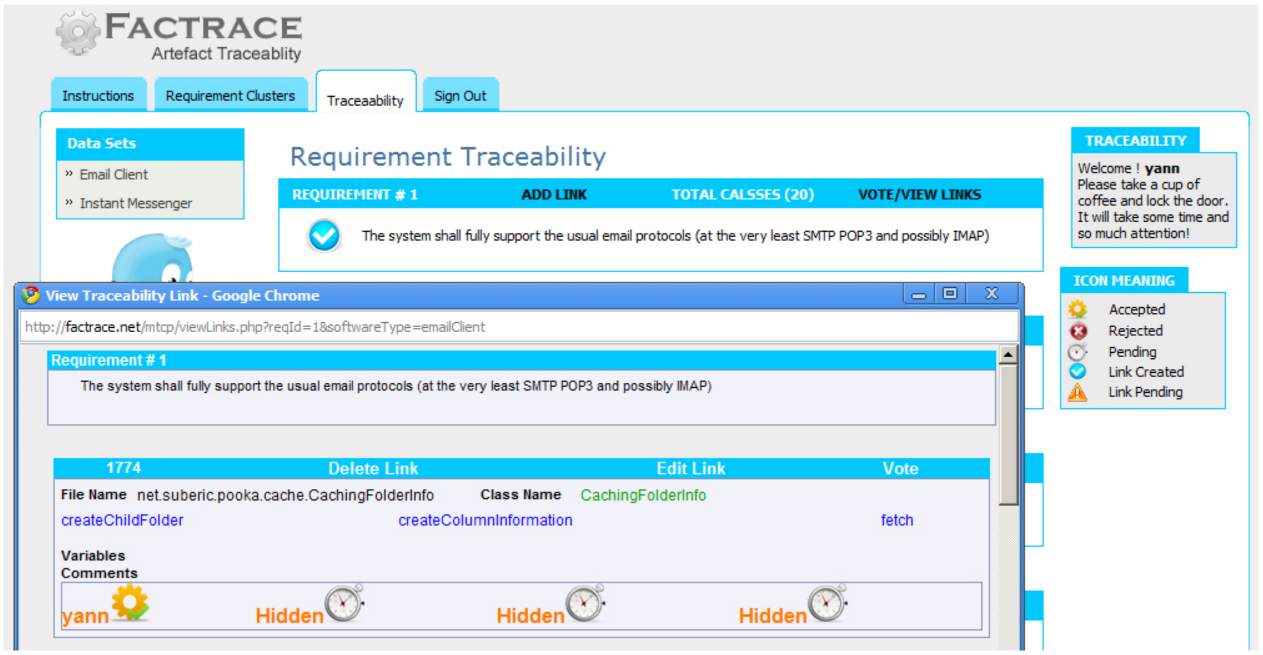
Requirement Elicitation: FacTrace currently supports Limesurvey[2] to gather requirements. Customers' requirements will automatically be stored in a database. Customers can write or upload their requirements in FacTrace specified XML format. FacTrace performs similarity analysis for all the elicited requirements. Similarity analysis helps developers to see if a single customer or multiple customers wrote the same requirement twice. It helps to remove the duplicated requirements. The similarity analysis is backed-up by clustering approaches, *i.e.*, agglomerative hierarchical clustering (Day and Edelsbrunner (1984)), to group similar requirements in a single cluster.

Requirement Analysis: FacTrace enables users to perform requirement analysis in few mouse clicks. A developer can see all similar requirements and label them. She can write a label or simply click on a specific requirement; it will automatically be placed in a text area to save time. In addition, she can also categorise each requirement as functional, non-functional, or an outlier. She can mark a requirement as duplicate if she thinks the current requirement is somehow semantically similar to another requirement. If any requirement has not been negotiated with a customer, developer can delete that requirement during requirement analysis.

Traceability Management: FacTrace helps in recovering traceability links between different software artifacts. For example, traceability links among requirements, source code, and CVS/SVN change logs. FacTrace allows experts to create new manual traceability links as well. It supports different level of granularity for creating traceability links. A developer can write a description for each link and other identifiers specifications as well.

Traceability Links Verification: To avoid bias when recovering traceability links, Fac-

2. https://www.limesurvey.org



The system shall fully support the usual email protocols (at the very least SMTP POP3 and possibly IMAP)

Welcome ! **yann**
Please take a cup of coffee and lock the door. It will take some time and so much attention!

http://factrace.net/mtcp/viewLinks.php?reqId=1&softwareType=emailClient

The system shall fully support the usual email protocols (at the very least SMTP POP3 and possibly IMAP)

File Name net.suberic.pooka.cache.CachingFolderInfo **Class Name** CachingFolderInfo

createChildFolder createColumnInformation fetch

Figure 1.2 Excerpt of Factrace GUI

Trace allows developers to vote on the recovered link by an IR technique and–or proposed approaches. It allows adding up to five developers for voting on each link. If three or more than three developers accept a link, then a link will be considered as a valid link by FacTrace. A developer can change their voting option at any time. All other developers' voting is hidden to avoid bias. She can see source code files in the source code viewer of FacTrace to verify each link.

Requirement Elicitation / Traceability Reports: FacTrace provides easy to understand tabular reports for requirement elicitation and traceability links. Reports can be exported in XML and CSV format. Reports are dynamically generate. All reports are updated as soon as a developer makes a change in a project.

1.6 Organisation of the Dissertation

The rest of this dissertation is organised as follows.

Chapter 2 – Background: This chapter presents the techniques and concepts that we use in this dissertation. The chapter starts by briefly describing the IR-based RT links recovery process. We explain the IR techniques that we use in this dissertation. The chapter continues with introducing different IR accuracy measurement to compare our proposed approaches with the state-of-the-art approaches. Next, it explains the statistical tests that we use to compare the accuracy of two given approaches.

Chapter 3 – Related Work: This chapter presents the concepts and research areas that are related to our research. The chapter starts by briefly presenting state-of-the-art traceability recovery approaches, Web trust model, document zoning, and the usage of eye-tracking system in software maintenance.

Chapter 4 – Creation of Experts: This chapter starts by describing how we could create multiple experts using existing sources of information. This chapter describes four more main approaches, *i.e.*, Histrace, LIBCROOS, COPARVO, and developer's knowledge, to create experts.

Chapter 5 – Combining Experts' Opinions: This chapter described a trust model, *i.e.*, Trumo, to combine different experts' opinion to reevaluate the trustworthiness of a link created by an IR technique.

Chapter 6 – Assessing Trustrace as a Traceability Recovery Method: This chapter explains that mining software repositories and using them as experts could yield better accuracy for RT. The chapter explains the mathematical model of the proposed solution for RT. It provides the details on experimental design and comparison of proposed approaches to the existing IR techniques.

Chapter 7 – Assessing the Usefulness of Trust Model: This chapter presents a different maintenance application, *i.e.*, bug location, of the proposed Trumo model (see Chapter 5). The chapter starts by explaining the usage of binary class relationships as experts to improve the accuracy of IR techniques for bug location. The chapter details on the slightly modified version of Trustrace for bug location and presents empirical study. The chapter continues with a discussion on the results and proposed approach for bug location.

Chapter 8 – Implementation of Coparvo-based Requirements Traceability: This chapter presents the concept of partition source code and using them as experts to vote on the traceability links recovered by an IR technique. The chapter starts by presenting the proposed approach, *i.e.*, COPARVO. It continues with the empirical experimental comparison of COPARVO with an IR technique. The chapter concludes with the discussion on results achieved with the COPARVO.

Chapter 9 – Using Developers' Knowledge for Improving Term Weighting Scheme: This chapter presents an empirical study with two controlled experiment on RT. The first part of the chapter provides the details on the first experiment with human subject. It continues with the experiment design, results, and findings of the experiments. The second part of the chapter explains a new weighting scheme based on the results achieved in first experiment. The chapter continues by providing details on second experiment. The chapter concludes based on the results and discussion of both experiments.

Chapter 10 – Conclusion and Future Work: This chapter revisits the main thesis and contributions of this dissertation. The chapter continues by describing potential opportunities for future research.

Appendix A: It provides the brief description of the data sets we used in this dissertation. It also provides the details on the procedure of creating manual oracles.

Appendix B: It provides the list of publication published during the time of my Ph.D.

Part I

Background

CHAPTER 2

INFORMATION RETRIEVAL PROCESS

This chapter provides the details of the three main techniques, *i.e.*, IR techniques, IR performance measures, and statistical tests, which we used in this dissertation. First, we briefly explain IR-based RT recovery process and IR techniques, *i.e.*, VSM, LSI, LDA, and JSM. Second, we explain the IR performance measures, *i.e.*, precision, recall, and F-Measure, which we use to compare the accuracy of an approach with some existing IR techniques. Third, we explain the statistical tests that we use to assess the improvement of proposed approaches.

2.1 IR Process

IR-based RTAs process is typically divided into three main steps (Antoniol *et al.* (2002b)). Figure 2.1 shows the high-level diagram of IR-based RT links recovery process. First, all the textual information contained in the requirements and source code is extracted and pre-processed by splitting terms, removing stop words and remaining words are then stemmed to its grammatical root. Second, all the stemmed terms are weighted using a term weighting scheme, *e.g.*, term frequency and inverse document frequency. Last, an IR technique computes the similarity between requirements and source code documents. Lastly, it generates a ranked list of potential traceability links. A high similarity between two documents shows a potential semantic link between them. Below we will explain each step in details:

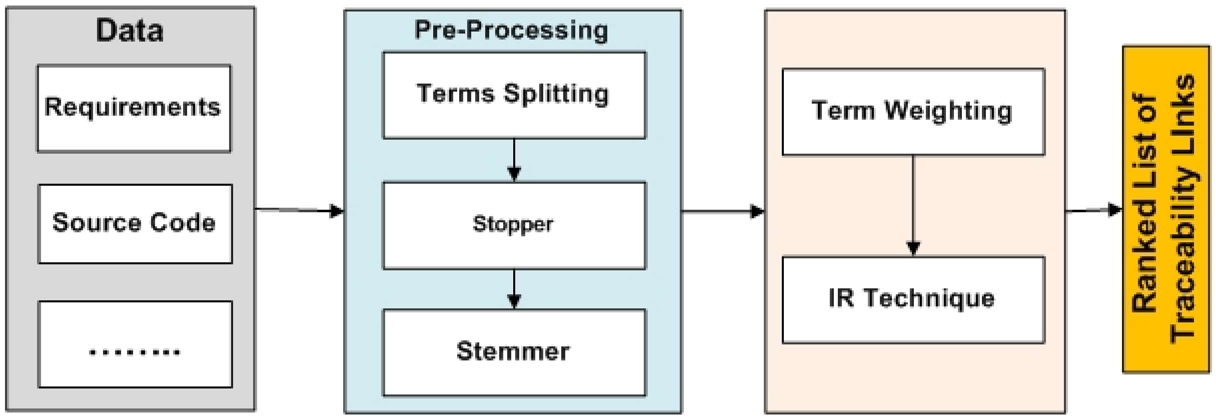


Figure 2.1 IR-based RT Links Recovery Process

2.1.1 Pre-Processing

To create traceability links, we extract all the identifiers[1] from source code and terms from requirements. In this dissertation, we use some IR techniques as an engine to create links between requirements and source code. IR techniques assume that all documents are in textual format. To extract source code identifiers, we use a source code parser, *e.g.*, a Java parser. The parser discards extra information, *e.g.*, primitive data types and language specific keywords, from the source code (Ali *et al.* (2011a)) and provides only identifier names. The extraction of the identifiers and terms is followed by a filtering, stopper, and stemmer process.

First, a text normalisation step converts all upper-case letters into lower-case letters. This step removes non-textual, *i.e.*, some numbers, mathematical symbols, brackets, etc., information and extra white spaces, from the documents. Some identifiers/terms could be combined with some special characters, *e.g.*, under_score, and–or CamelCase naming convention. Therefore, we split all the joined terms to make them separate terms. For example, SendMessage and send_message are split into the terms "send message".

Second, the input of this step is normalised text that could contain some common words, *e.g.*, articles, punctuation, etc. These common words are considered as noise in the text because it does not represent semantics of a document. Thus, in this step, we use a stop word list to remove all the stop words. In this dissertation, we used English language stop words list as all our documents are in English.

The third step is stemming. An English stemmer, for example, would identify the terms "dog", "dogs" and–or "doggy" as based on the root "dog". In this dissertation, we use the Porter stemmer (Porter (1997)). An IR technique computes the similarity between two documents based on similar terms in both documents. However, due to different postfix, IR techniques would consider them, *e.g.*, access, accessed, as two different terms and it would result into low similarity between two documents. Thus, it becomes important to perform morphological analysis to convert plural into singular and to bring back inflected forms to their morphemes.

2.1.2 Existing Term Weighting

An IR technique converts all the documents into vectors to compute the similarities among them. To convert documents terms into vectors, each term is assigned a weight. Various schemes for weighting terms have been proposed in literature (Abadi *et al.* (2008); Antoniol *et al.* (2002b); of Dayton Research Institute (1963)). Widely used weighting schemes are characterised in two main categories: probabilistic (of Dayton Research Institute (1963)) and

1. In the following, we use term identifiers to refer all source code entities, *i.e.*, class name, method name, variable name, and comments.

algebraic models (Antoniol *et al.* (2002b)). Probabilistic weighting models heavily depends on probability estimations of terms and algebraic weighting models depends on terms distribution in a document and–or whole corpora. However, in both weighting schemes following two main factors are considered important:

– **Term frequency (TF)**: TF is often called local frequency. If a term appears multiple times in a document then it would be assigned higher TF than the others.
– **Global frequency (GF)**: If a word appears in multiple documents then the term is not considered representative of documents content. The global frequency is also called inverse document frequency (IDF).

Early IR techniques used TF to link two documents. If a term appears multiple times in a single or multiple documents then IR technique would recommend that document as relevant document to a query. However, multiple occurrences of a term do not show that it is important term. Jones (1972) proposed IDF to reduce the weight of a term if a term appears in multiple documents. TF is calculated as:

$$TF = \frac{n_{i,j}}{\sum_k n_{k,j}}$$

where $n_{i,j}$ is the occurrences of a term t_i in document d_j and $\sum_k n_{k,j}$ is the sum of the occurrences of all the terms in document d_j.

The IDF of a term is computed as:

$$IDF = \log\left(\frac{|D|}{|d : t_i \in d|}\right)$$

where $|D|$ is the total number of documents d in the corpus, and $|d : t_i \in d|$ is the number of documents in which the term t_i appears. In this dissertation, we use TF/IDF weighting scheme.

2.2 IR Techniques

In this dissertation, to build sets of traceability links, we use some IR techniques, in particular VSM(Antoniol *et al.* (2002b)), LSI(Marcus *et al.* (2003)), and JSM (Abadi *et al.* (2008)). To identify concepts in the source code, we use LDA (Asuncion *et al.* (2010)). Abadi *et al.* (2008) performed experiments using different IR techniques to recover traceability links. Their results show that the VSM and the JSM outperform other IR techniques. In addition, these two techniques do not depend on any experimental value for tuning on some particular dataset. Thus, in this dissertation, we use JSM and VSM to recover traceability links in most of our experiments. Both techniques essentially use term-by-document matrices.

Consequently, we choose the well-known TF/IDF weighting scheme (Antoniol *et al.* (2002b)) for the VSM and the normalised term frequency measure (Abadi *et al.* (2008)) for the JSM. These two measures and IR techniques are state-of-the-art for traceability. In the following, we explain all the techniques and weighting in details.

2.2.1 Vector Space Model

Many traceability links recovery techniques (Antoniol *et al.* (2002b); Baeza-Yates and Ribeiro-Neto (1999a); Lucia *et al.* (2007)) use VSM as baseline algorithm. In a VSM, documents are represented as vector in the space of all the terms. Various term weighting schemes can be used to construct these vectors. If a term belongs to a document then it gets a non-zero value in the VSM along the dimension corresponding to the term. A document collection in VSM is represented by a term by document matrix, *i.e.*, $m \in n$ matrix, where m is the number of terms and n is the number of documents in the corpus.

Once documents are represented as vectors of terms in a VSM, traceability links are created between every two documents, *e.g.*, a requirement and a source code class, with different similarity value depending on each pair of documents. The similarity value between two documents is measured by the cosine of the angle between their corresponding vectors. Cosine values are in $[-1, 1]$ but negative values are discarded and a link has thus a value in $]0, 1]$ because similarity cannot be negative and zero between two documents. Finally, the ranked list of recovered links and a similarity threshold are used to create a set of candidate links to be manually verified (Antoniol *et al.* (2002b)). The angle between two vectors is used as a measure of divergence between the vectors. If R is a requirement vector and C is a source code vector, then the similarity of requirement to source code can be calculated as follows (Baeza-Yates and Ribeiro-Neto (1999a)):

$$sim(R, C) = \frac{R \cdot C}{||R|| \cdot ||C||} = \frac{\sum_{t_i \in R} w_{t_{iR}} \cdot \sum_{t_i \in C} w_{t_{iC}}}{\sqrt{\sum_{t_i \in R} w_{t_{iR}}^2} \cdot \sqrt{\sum_{t_i \in C} w_{t_{iC}}^2}}$$

where $w_{t_{iR}}$ is the weight of the i^{th} term in the query vector R, and $w_{t_{iC}}$ is the weight of the i^{th} term in the query vector C. Smaller the vector angle is, higher is the similarity between two documents.

2.2.2 Latent Semantic Indexing

VSM has a limitation, it does not address the synonymy and polysemy problems and relations between terms (Deerwester *et al.* (1990)). For example, having a term "home" in one document and “house” in another document results in non-similar documents. LSI takes into account the association between terms and documents to overcome synonymy and polysemy problems. LSI assumes that there is an underlying latent structure in word usage for every document set (Deerwester *et al.* (1990)). The processed corpus is transformed into a term-by-document ($m \in n$) matrix A, where each document is represented as a vector of terms. The values of the matrix cells represent the weights of the terms, which could be computed using the traditional *TF/IDF* weighting schemes.

The matrix is then decomposed, using *Singular Value Decomposition (SVD)* (Deerwester *et al.* (1990)), into the product of three other matrices:

$$A = U \times S \times V$$

where U is the $m \times r$ matrix of the terms (orthogonal columns) containing the left singular vectors, V is the $r \times n$ matrix of the documents (orthogonal columns) containing the right singular vectors, S is an $r \times r$ diagonal matrix of singular values, and r is the rank of A. To reduce the matrix size, all the singular values in S are ordered by size. All the values after the first largest k value could be set to zero. Thus, deleting the zero rows and columns of S and corresponding columns of U and rows of V would produce the following reduced matrix:

$$A_k = U_k \times S_k \times V_k$$

where the matrix A_k is approximately equal to A and is of rank $k < r$. The choice of k value, *i.e.*, the SVD reduction of the latent structure, is critical and still an open issue in the natural language processing literature (Deerwester *et al.* (1990); Marcus *et al.* (2003)). We want a value of k that is large enough to fit all the real structures in the data but small enough so we do not also fit the sampling error or unimportant details in the data.

2.2.3 Latent Dirichlet Allocation

Automatically discovering the core concepts of data has stimulated the development of dimensionality reduction techniques, *e.g.*, LSI. Hofmann proposed probabilistic version of LSI (Hofmann (1999)) (PLSI) that has a more solid statistical foundation than LSI, because it is based on the likelihood principle and defines a proper generative model of the data. Blei *et al.* (2003) proposed a fully generative Bayesian model known as LDA. LDA overcomes the PLSI issues, *e.g.*, over-fitting, and achieves better results than PLSI (Hofmann (2001)). Thus, in this dissertation, we use LDA to discover the core concepts of the data.

LDA is an unsupervised machine learning technique that does not require any training data to train itself. The only required input to LDA is some tuning parameters. LDA considers that documents are represented as a mixture of words acquired from different latent topics, where each topic T is characterised by a distribution of words W.

In particular, LDA takes the following inputs:

– D, documents;

– k, number of topics;

– α, Dirichlet hyperparameter for topics' proportions;

– β, Dirichlet hyperparameter for topics' multinomials;

A term-topic probability distribution (ϕ) is drawn from Dirichlet distribution with hyper-parameter β. A document D is associated with the topic-document probability distribution Θ drawn from a dirichlet with hyperparameter α. Hyperparameter values typically are set according to the de-facto standard heuristics: $\alpha = 50/k$ and $\beta = 0.01$ or $\beta = 0.1$ (Griffiths and Steyvers (2004); Wei and Croft (2006)). Biggers *et al.* (2012) performed an extensive study on feature location using LDA. They analysed the impact of different parameters of LDA on feature location. Their results show, which LDA parameters would be good for different size of datasets.

2.2.4 Jensen-Shannon Divergence Model

The JSM is an IR technique proposed by Abadi *et al.* (2008). It is driven by a probabilistic approach and hypothesis testing technique. JSM represents each document through a probability distribution, *i.e.*, normalised term-by-document matrix. The probability distribution of a document is:

$$pb_{i,j} = \frac{n(w,d)}{T_d}$$

where $n(w,d)$ is the number of times a word appears in a document d and T_d is the total number of words appearing in a document d. The empirical distribution can be modified to take into account the term's global weight, *e.g.*, IDF. After considering the global weight, each document distribution must be normalised as:

$$p_{i,j} = \frac{pb_{i,j} \cdot IDF_{i,j}}{\sum_{i=0}^{n} pb_{i,j} \cdot IDF_{i,j}}$$

where pb_j and IDF_j is the probability distribution and inverse document frequency of i^{th} term in j^{th} document, respectively.

Once the documents are represented as probability distribution, JSM computes the distance between two documents' probability distribution and returns a ranked list of traceability links. JSM ranks source documents, *e.g.*, requirements, via the "distance" of their probability distributions to that of the target documents, *e.g.*, source code:

$$\begin{aligned} JSM(q,d) &= H\left(\frac{p_q + p_d}{2}\right) - \frac{H(p_q) + H(p_d)}{2} \\ H(p) &= \sum h(p(w)) \\ h(x) &= -x \log x \end{aligned}$$

where $H(p)$ is the entropy of the probability distribution p, and p_q and p_d are the probability distributions of the two documents (a "query" and a "document"), respectively. By definition, $h(0) \equiv 0$. We compute the similarity between two documents using $1 - JSM(q,d)$. The similarity values are in $]0,1]$.

2.3 Generation of Traceability Links' Sets

To evaluate the effectiveness of two RTAs, we generate various traceability links' sets at different thresholds. We then use these sets to compute precision, recall, and–or F-Measure values. These sets help us to evaluate, which approach is better than the other at all the threshold values or some specific thresholds values. We perform several experiments with different threshold values on the recovered links, by two RTAs, to perform statistical tests. In literature following three main threshold strategies have been proposed by researchers:

Scale threshold: It is computed as the percentage of the maximum similarity value between two software artifacts, where threshold t is $0 \leq t \leq 1$ (Antoniol *et al.* (2002b)). In this case, the higher the value of the threshold t, the smaller the set of links returned by a query.

Constant threshold: It (Marcus and Maletic (2003b)) has values between $[0,1]$; a good and widely used threshold is $t = 0.7$. However, if the maximum similarity between two software artifacts is less than 0.7 then this threshold $t = 0.7$ would not be suitable.

Variable threshold: This is an extension of the constant threshold approach (De Lucia *et al.* (2004)). When using a variable threshold, the constant threshold is projected onto a particular interval, where the lower bound is the minimum similarity and upper bound is the maximum similarity between two software artifacts. Thus, the variable threshold has values between 0% to 100% and on the basis of this value the method determines a cosine threshold.

In this dissertation, we use scale threshold. We use a threshold t to prune the set of traceability links, keeping only links whose similarities values are greater than or equal to $t \in]0,1]$. We use different values of t from 0.01 to 1 per step of 0.01 to obtain different sets of traceability links with varying precision, recall, and–or F-measure values, for all approaches.

2.4 IR Performance Measures

Now we describe the IR metrics that we use to compute the accuracy of each set using IR-based metrics.

2.4.1 Precision and Recall

We use two well-known IR metrics, precision and recall, to evaluate the accuracy of our experiment results. Both measures have values in the interval $[0, 1]$. Precision and recall values are calculated for all the traceability links retrieved above a threshold. A developer could define threshold value based on the project scope and–or retrieved documents.

$$Precision = \frac{|\{relevant\ documents\} \cap \{retrieved\ documents\}|}{|\{retrieved\ documents\}|}$$

Precision is defined as the total number of relevant documents retrieved divided by the total number of retrieved documents by an approach. Precision considers all retrieved documents above than the threshold value. This measure is called precision at n or $P@n$. If the value is 1 for precision it means that all the recovered documents are correct.

$$Recall = \frac{|\{relevant\ documents\} \cap \{retrieved\ documents\}|}{|\{relevant\ documents\}|}$$

Recall is defined as the relevant documents retrieved divided by the total number of relevant documents. Documents could be a query or result of query execution. It is ratio between the number of documents that are successfully retrieved and the number of documents that should be retrieved. If the value is 1 for recall, it means all relevant documents have been retrieved.

2.4.2 F-Measure

The precision and recall are two independent metrics to measure two different accuracy concepts. F-measure is the harmonic mean of precision and recall that is computed as:

$$F = 2 \times \frac{P \times R}{P + R}$$

where P is the precision, R is the recall of retrieved documents and F is the harmonic mean of P and R. This F-measure is also known as F_1 measure. In F_1 measure, precision and recall are equally weighted.

The function F assumes values in the interval $[0, 1]$. It is 0 when no relevant documents have been retrieved and is 1 when all retrieved documents are relevant. Further, the harmonic mean F assumes a high value only when both recall and precision are high. Therefore, determination of the maximum value for F can be interpreted as an attempt to find the best possible compromise between recall and precision (Baeza-Yates and Ribeiro-Neto (1999b)).

The generic formula of F-measure is:

$$F_\beta \;=\; (1+\beta) \times \frac{P \times R}{\beta \times P + R}$$

where β is a parameter that can be tuned to weight more precision over recall or vice versa. For example, F_2 weights recall twice as much as precision and $F_{0.5}$ weights precision twice as much as recall.

2.5 Statistical Hypothesis Testing

In this dissertation, to evaluate the effectiveness of an approach and measure the improvement brought by the approach, we perform empirical experiments. We use the IR metrics, *e.g.*, precision and recall, to measure any improvement. The results could be improved on an average at certain threshold points. However, average does not give much insight of the actual improvement. Statistical tests provide the in depth analysis of data points to measure the improvement.

To perform statistical test, first, we pose a null hypothesis, *e.g.*, there is no difference in the recall of the recovered traceability links when using LSI or VSM, that we want to reject. To reject a null hypothesis, we define a significance level of a test, *i.e.*, α. It is an upper bound of the probability for rejecting the null hypothesis. We reject a null hypothesis if the result value of a statistical test is below the significance level, *e.g.*, 0.05. We accept alternate hypothesis, *e.g.*, there is a difference in the recall of recovered links when using LSI or VSM, or provide an explanation if we do not reject null hypothesis. Second, to select an appropriate statistical test we analyse the distribution of data points. Lastly, we perform a statistical test to get a probability value, *i.e.*, p-value, to verify our hypothesis. p-value is compared against the significance level. We reject null the hypothesis if the p-value is less than significance level.

In this dissertation, we perform paired-statistical tests to measure the improvements brought by an approach over an existing approach. In paired-statistical test, two chosen approaches must have the same number of data points on the same subjects. Therefore, we use the same threshold t value for both approaches. For example, if we want to compare VSM and LSI then if VSM discards all traceability links whose textual similarity values are below than the 0.83 threshold, then we also use the same 0.83 as upper threshold for LSI. Then, we assess whether the differences in precision, recall, and–or F-measure values, in function of t (see Section 2.3), are statistically significant between the two approaches.

Following we provide the details on creating the traceability links sets and using them to evaluate the effectiveness of proposed approaches.

2.5.1 Statistical Tests

We perform appropriate statistical tests to analyse whether the improvement in accuracy with proposed approach is indeed an improvement or it is by chance. In the following, we discuss the statistical tests we use in this dissertation.

Shipro-Wilk Test:

There are two types of data, *i.e.*, normally distributed and any other distribution, and two types of statistical analysis, *i.e.*, parametric and non-parametric tests. Parametric tests are for normally distributed data and nonparametric tests are for any other data distribution. Pre-requisite to perform/select any statistical test is to assess the normality of the data distribution. In this dissertation, we use Shipro-Wilk test (Shapiro and Wilk (1965)) to analyse the distribution of the data. The Shapiro-Wilk test calculates whether a random sample, *e.g.*, $S_1, S_2 \ldots S_n$ comes from a normal distribution.

Mann-Whitney Test:

The Mann-Whitney (Wohlin *et al.* (2000)) is also known as the Wilcoxon Rank sum test; because it is directly related to the sum of ranks. Mann-Whitney assesses how many times a set Y precedes a set X in two samples. It is a non-parametric test and an alternative to the two-sample student's t-test. Mann-Whitney is a robust statistical test that could also be used for small sample sizes, *e.g.*, 5 to 20 samples. It could also be used when the sample values are captured using an arbitrary scale which cannot be measured accurately.

Kruskal-Wallis Test:

The Kruskal-Wallis rank sum test (Wohlin *et al.* (2000)) is a non-parametric method for testing the equality of the population medians among different groups. It is performed on ranked data, so the measurement observations are converted to their ranks in ascending order. The loss of information involved in replacing original values with their ranks can make Kruskal-Wallis test a less powerful test than an ANOVA test (Wohlin *et al.* (2000)). Thus, if the data is normally distributed, a developer should use ANOVA test.

CHAPTER 3

RELATED WORK

Traceability recovery, feature location, trust models, and eye-tracker topics are related to this research work. At the end of the chapter, we provide a summary of all the existing techniques and compare if they have any similar functionalities like proposed in this dissertation.

3.1 Traceability Approaches

Traceability approaches could be divided into three main categories, *i.e.*, dynamic , static, and hybrid. **Dynamic traceability approaches** (Liu *et al.* (2007)) require a system to be compilable and executable to perform traceability creation tasks. It also requires pre-defined scenarios to execute the software system. Dynamic approaches collect and analyse execution traces (Wilde and Casey (1996)) to identify which methods a software system is executing for a specific scenario. However, it doesn't help to distinguish between overlapping scenarios, because a single method could participate in several scenarios. More importantly, due to bugs and–or some other issues a legacy system may not be executable. Thus, it may not be possible to collect execution traces.

Static traceability approaches (Abadi *et al.* (2008); Antoniol *et al.* (2002a); Marcus and Maletic (2003c)) use source code structure and–or textual information to recover traceability links between high-level and low-level software artifacts. Static traceability approaches have received much attention over the past decade in the scientific literature. Antoniol *et al.* (2002a) used IR-based probabilistic models and VSM to link textual documents. Antoniol *et al.* (2000b) discussed how a traceability recovery tool based on the probabilistic model can improve the retrieval performances by learning from user feedbacks. Marcus and Maletic (2003c) used LSI to perform the same case studies as in (Antoniol *et al.* (2002a)) and compared the performances of LSI with respect to the VSM and probabilistic models. Their results showed that LSI could provide better performance without the need of a stemmer that is required for the VSM and probabilistic models. Zou *et al.* (2010) performed empirical studies to investigate query term coverage, phrasing, and project glossary term-based enhancement methods. These methods are designed to improve the performance of automated tracing tool based on a probabilistic model. The authors proposed a procedure to automatically extract critical keywords and phrases from a set of traceable artifacts to enhance the automated trace retrieval.

Abadi *et al.* (2008); Oliveto *et al.* (2010) compared different IR techniques, *i.e.*, VSM, LSI, LDA, and JSM. Their results showed that VSM and JSM provide better results than the other approaches. Oliveto *et al.* (2010) demonstrated that LDA could recover some links that other IR-based RTAs miss. However, over all accuracy of LDA was not satisfactory. Gethers *et al.* (2011) proposed an integrated approach to orthogonally combine IR techniques, *i.e.*, VSM, JSM, and relational topic modelling, which have been statistically shown to produce dissimilar results. Their proposed approach uses each IR technique as an expert and uses PCA-based weighting scheme to combine them. De Lucia *et al.* (2011) proposed the use of smoothing filters to reduce the effect of noise in software artifacts and improve the performances of traceability recovery methods. The authors conducted an empirical study on two datasets, *i.e.*, EasyClinic and Pine. Their results showed that the usage of a smoothing filter can significantly improve the performances of VSM and LSI.

Some researchers (Antoniol *et al.* (2001, 2000a); McMillan *et al.* (2009)) have explored the source code structure and how to combine it with source code textual information to improve the accuracy of IR techniques. Antoniol *et al.* (2001, 2000a) proposed an approach for automatically recovering traceability links between OO design models and source code. The authors used class attributes as traceability anchors to recover traceability links. McMillan *et al.* (2009) proposed a technique for indirectly recovering traceability links by combining textual with structural information. The authors conjectured that related requirements share related source code elements. Their case study showed that their combined approach improves the precision and recall as compared to stand-alone methods based solely on analysing textual similarities. Grechanik *et al.* (2007) proposed a novel approach for automating part of the process of recovering traceability links between types and variables in Java programs and elements of use-case diagrams. The authors evaluated their prototype implementation on open-source and commercial software, and their results suggested that their approach can recover many traceability links with a high degree of automation and precision.

Software repositories have been explored by many researchers (Kagdi *et al.* (2007); Kagdi and Maletic (2007)) to recover traceability links. Kagdi *et al.* (2007) presented a heuristic-based approach to recover traceability links between software artifacts using software systems' version history. Their approach assumes, two files could have a potential link between them if they co-change(Kagdi and Maletic (2007)). However, it is quite possible that two files are co-changing but they do not have any semantic relationship. It is also likely that some software artifacts do not have software repositories and, in such a case, their approach cannot find link from/to these documents, *e.g.*, requirement specifications. In addition, their approach does not analyse contents of the CVS/SVN commit logs and files that were committed in

CVS/SVN. More importantly, in co-change-based traceability (Kagdi *et al.* (2007); Kagdi and Maletic (2007); Kagdi *et al.* (2006)), if two or more files have a link but they were not co-changed then these approaches fail to find a link. Our proposed novel approach, *i.e.*, Trustrace, is not dependent on co-changes and overcomes these limitations.

The importance of a term position in different textual documents have been explored by various researchers (Kowalski (2010); Erol *et al.* (2006); Sun *et al.* (2004)). They observed that if a term appears in different zones of a document, then its importance changes. This idea that a term has different importance to a reader depending on where it appears has been investigated in the domain of IR (Kowalski (2010)). Search engines, such as Google[1], assign higher ranks to the Web pages that contain the searched terms in specific parts of a page, *e.g.*, title and body. However, this phenomenon is not evaluated on source code. Erol *et al.* (2006) used a questionnaire to ask participants which parts of a document are more important than the others. They concluded that titles, figures, and abstracts are the most important parts for both searching and understanding documents while figure caption is only important for understanding. Physically dividing documents into zones, *e.g.*, title and abstract, has been already investigated in the field of IR. However, in this dissertation, we report the first analysis of developers' visual attention on SCEs using eye-tracking. In particular, we believe that people's preferences for documents may differ from developers' preferences for source code. In addition, conceptual division of a source code file, *e.g.*, application and domain concepts, is not being studied yet.

The precision and recall (Antoniol *et al.* (2002b)) of the RT slinks recovered during traceability analyses are influenced by a variety of factors, including the semantic distance between high-level documentation and low-level artifacts, and the way in which queries are formulated. Different IR techniques have different way to compute the similarity among documents. Some researchers, *e.g.*, Lucia *et al.* (2007) and Abadi *et al.* (2008), have performed empirical studies to analyse which IR technique works better than the other. On several dataset, the VSM and JSM perform favourably in comparison to more complex techniques, such as LSI (Abadi *et al.* (2008)) and LDA (Asuncion *et al.* (2010)). Yet, algebraic model, *e.g.*, VSM (Antoniol *et al.* (2002b)) and probabilistic model, *e.g.*, JSM (Abadi *et al.* (2008)), are a reference baseline for both feature location Poshyvanyk *et al.* (2007); Zhao *et al.* (2006a) and traceability recovery Antoniol *et al.* (2002b); Lucia *et al.* (2007).

Hybrid traceability approaches (Poshyvanyk *et al.* (2007); Dit *et al.* (2011b); Eaddy *et al.* (2008b)) combine static and dynamic information. Poshyvanyk *et al.* (2007) combined a scenario-based probabilistic ranking of events and an IR technique that uses LSI for feature location. Their empirical study shows that combing dynamic and static information can per-

1. www.google.com

form better than a single IR technique. Eaddy *et al.* (2008b) proposed a new technique called prune-dependency analysis that can be combined with existing techniques to dramatically improve the accuracy of concern location. The authors developed CERBERUS, a hybrid technique for concern location that combines IR techniques, execution tracing, and prune dependency analysis. The results achieved with hybrid approaches show that dynamic data helps to improve the accuracy of static traceability approaches. However, as we previously discussed that it may not be easy to collect dynamic traces data due to bugs and some other issues. Thus, hybrid approaches face the same problems and limitations as dynamic approaches.

The results (Abadi *et al.* (2008); Antoniol *et al.* (2002a); Marcus and Maletic (2003c); Oliveto *et al.* (2010); Gethers *et al.* (2011)) achieved by static approaches showed that they do not require an executable software system. Thus, static traceability approaches could be applied to a system that contains a bug or is not executable. In this dissertation, we propose static traceability approaches and compare them with existing state-of-the-art static approaches.

3.2 Feature Location

Bug or feature location is a search activity, whether a developer searches the source code to find the classes that are playing a role to implement a feature or causing a bug. In the search results, developers tend to look at the top few results only (Poshyvanyk *et al.* (2007)). Many researchers have proposed automated and semi-automated approaches to facilitate developers to locate a feature or a bug. Similar to RTAs, all feature location approaches could also be divided into three categories, dynamic, static, and hybrid.

Static analyses (Robillard (2008)), execution traces (Wilde and Casey (1996)), and IR techniques (Antoniol *et al.* (2002b); Marcus and Maletic (2003a)) have been used by researchers since the early work on feature location Wilde and Casey (1996). Often, IR techniques (Antoniol *et al.* (2002b); Marcus and Maletic (2003a); Poshyvanyk *et al.* (2007)) use VSM, probabilistic rankings, or a transformed VSM matrix using LSI. Whenever available, dynamic data (Wilde and Casey (1996); Poshyvanyk *et al.* (2007)) proved to be complementary and useful for traceability recovery by reducing the search space. Recently, high-level documentation was mapped into source code features using a variety of information sources (Poshyvanyk *et al.* (2007)). Marcus and Maletic (2003c) proposed a LSI-based approach to locate features in the source code. Their approach allows developers to formulate queries in natural language and results are returned as a list of source code elements ranked by the relevance to the query.

Combining branch-reserving call graph (Zhao *et al.* (2006b)), execution traces (Poshyvanyk *et al.* (2007)), call graph (Shao and Smith (2009)), with IR techniques, and structural dependencies (Robillard (2008)) provide promising results. Zhao *et al.* (2006b) proposed SNIAFL, a static, non-interactive feature location approach. SNIAFL combines IR techniques with a branch-reserving call graph (BRCG) for feature location. They used VSM to generate an initial set of methods related to a feature, and BRCG to filter out false positives. Poshyvanyk *et al.* (2007) formulated the feature location problem as combination of the opinions of different experts. They used a scenario-based probabilistic ranking of event and an IR technique as experts to locate features in the source code. Shao and Smith (2009) combined IR and static control flow information for feature location. They used LSI to rank all the methods in a software system by their relevance to a query. Then, for each method in the ranked list, a call graph was constructed and assigned a call graph score. The call graph counts the method's direct neighbours that also appeared in LSI ranked list. Finally, they combined the LSI cosine similarity and call graph score to produce a new ranked list. Robillard (2008) proposed an approach that analyses the topology of structural dependencies in a software system to propose relevant program elements for the developers to investigate. It takes as input a set of program elements of interest to a developer and produces a fuzzy set describing other elements of potential interest.

3.3 Binary Class Relationships

OO programming is a method of implementation in which programs are organised as a collection of collaborative objects (Booch (1991)). As real world entities, classes in OO programs do not exist in isolation; they cooperate through the BCRs between them. The main BCRs are inheritance, association, aggregation, and using (use relation) (Booch (1991)). In this dissertation, we will consider these four BCRs. In OO programs, relationships are as important as the objects themselves (Rumbaugh (1987)). Class relationships allow classes to share data, to define more complex structures or to participate in the implementation of a program feature (Booch (1991); Purdum (2008)). Therefore, classes involved in the implementation of a feature (program behaviour) are probably linked by class relationships. Program behaviour that deviates from its specification is called a bug (Allen (2002)). Thus, to locate relevant classes involved in the occurrence of a bug is similar to locating classes involved in the implementation of a feature that has not been correctly implemented. Based on this observation, we believe that using information about BCRs to locate a bug can be helpful.

Although class relationships are essential in the implementation of features and for program comprehension tasks, they are not all explicit in the source code (Guéhéneuc and Albin-Amiot (2004); Pearce and Noble (2006)). It is not an obvious task to recover class relationships in source code. Indeed, many researchers propose various approaches (Jackson and Waingold (1999); Guéhéneuc and Albin-Amiot (2004)) to extract class relationships in the source code. Part of our approach is based on the approach proposed by Guéhéneuc and Albin-Amiot (2004). The authors formalised BCRs based on four independent-language properties: exclusivity, receiver type, life-time, and the number of instances. Using these properties and specific algorithms, they recovered class relationships in the source code of a software system. They provided this technique in the Ptidej tool suite[2] Guéhéneuc (2005).

To the best of our knowledge, none of previous work performed experiments to analyse what are the important BCRs, in particular at class-level, for bug location. In addition, how BCRs could be combined with IR techniques to improve the accuracy, in terms of ranking, for bug location. The work presented in this dissertation is complementary to the existing IR BLTs, because it exploits the BCRs of OO programs to improve the accuracy of IR techniques.

3.4 Web Trust Model

Our proposed novel approaches are influenced by the Web trust model (Berg and Van (2001); McKnight *et al.* (2002); Palmer *et al.* (2000); Koufaris and Hampton-Sosa (2004)). There are two type of trusts in e-commerce: first a customer's initial trust (Koufaris and Hampton-Sosa (2004)) when she interacts with a Website for the first time and, second, the Website reputation trust (Artza and Gil (2007)) that develops over time and after repeated experiences. When customers hesitate to buy things online, they may ask their friends, family, and other buyers to make sure that they can trust a Website. Many researchers investigated (Berg and Van (2001); Palmer *et al.* (2000); Koufaris and Hampton-Sosa (2004); Artza and Gil (2007); Grandison and Sloman (2000)) the problem of increasing customers' trust in a Website. Some researchers (Berg and Van (2001); Palmer *et al.* (2000)) suggested that using more sources of information can increase the trust in a Website. Thus, our main conjecture in this dissertation is similar to Web trust model that using more sources of information could yield better accuracy, in other words more trust of true positive links, for IR-based RTAs. Our proposed approaches use traceability links from requirement to source code as initial trust and then uses CVS/SVN commit logs, bug reports, mailing lists, and so on, as reputation trust for a traceability link. As the reputation of a link increases, the trust in this link also increases. Below we briefly explain the available Web trust models:

2. http://www.ptidej.net/download

Berg and Van (2001) attempted to develop the equivalent of symbolons, for e-commerce. Their study deals with a specific type of electronic medium, the World Wide Web. In particular, it focuses on the so-called Web assurance services, which provide certification of legitimacy of Websites. The authors highlighted business-to-consumer commerce, but also pay attention to the challenges regarding Web assurance services in the business-to-business environment.

Palmer *et al.* (2000) presented an empirical investigation of how firms can improve customers' trust by exploring and using Trusted Third Parties (TTPs) and privacy statements. Their exploratory data showed that the use of TTPs and privacy statements increase a customer's trust in a Website. Wanga *et al.* (2010) presented a novel content trust learning algorithm that use the content of a Website as a trust point to distinguish trustable Web contents and spam contents.

McKnight *et al.* (2002) empirically tested the factors that may influence initial trust in a Web-based company. The authors tested a trust building model for new customers of a fictitious legal advice Website and found that perceived company reputation and perceived Website quality both had a significant positive relationship with initial trust with the company.

Koufaris and Hampton-Sosa (2004) proposed a model to explain how new customers of a Web-based company develop initial trust in the company after their first visit. The authors empirically tested using a questionnaire-based field study. The results indicate that perceived company reputation and willingness to customise products and services can significantly affect initial trust.

3.5 Eye-Tracking

Eye-trackers have recently been used to study program comprehension. De Smet *et al.* (2011) performed three different eye-tracking experiments to investigate the impact of Visitor, Composite and Observer design patterns, and Model View Controller style on comprehension tasks. Yusuf *et al.* (2007) also conducted a study using an eye-tracker to analyse how well a developer comprehends UML class diagrams. Their results showed that developers tend to use stereotypes, colours, and layout to have a more efficient exploration and navigation of the class diagrams.

Bednarik and Tukiainen (2006a) used eye-tracking systems to characterise the program comprehension strategies deployed by developers during dynamic program visualisation. Uwano *et al.* (2006) also conducted an experiment to characterise the performance of developers while reviewing the source code. They concluded that the more developers read the source code, the more efficiently they find defects. Sharif and Maletic (2010) carried an eye-tracking experiment to analyse the effect of identifier style, *i.e.*, camel case and underscore, on developers' performance while reading source code. Sharif and Kagdi (2011) suggested in their position paper that eye-tracking system could be used in the field of traceability.

To the best of our knowledge, none of this previous work performed experiment to analyse what are important SCEs for developers, how does SCE impact RTAs if different importance are given to each SCE, and the role of domain and implementation entities. The work presented in this dissertation is complementary to the existing IR-based RTAs, because it exploits the identifiers' importance based on their position (*e.g.*, class or method names) or their role (*e.g.*, domain or implementation).

3.6 Summary of Related Approaches

Table 3.1 summarises the related work on traceability recovery approaches, eye-tracker, and Web trust model. All these research fields are closely related to the work presented in this dissertation. The column external information in Table 3.1 shows whether the approach uses any external information, *e.g.*, execution traces, software repositories, human knowledge, or not. Multiple expert column shows if the current approach accommodates more than one expert opinion. CTW, SCP, and BCRS column shows that if an approach support customised term weighting, source code partitioning, and binary class relationships respectively. The automated weights column shows if the approach is capable of automatically assigning weights to each expert. Only one approach (Gethers *et al.* (2011)) provides automated support for assigning weights to multiple experts. The column FL and RT shows if current approach support feature location or requirements traceability. Gethers *et al.* (2011) approach currently supports FL only and have not been applied on RT yet. ET and WTM column shows that if current approach supports/uses Web trust model and–or eye-tracker. The work presented in this dissertation is complementary to existing IR-based techniques, because it uses current state-of-the-art technique to create baseline links and uses more sources of information as experts to filter out false-positive links and–or increase the trust over remaining links. Table 3.1 shows the work presented in this dissertation is novel.

Table 3.1 Related work summary of closely related approaches to the work presented in this dissertation. CTW represents customised term weighting, SCP represents source code partitioning, Mul. Exp. represents the option to add multiple experts, AEW represents automated expert weighting,ET, WTM supp., Fl, and RT represent eye-tracker, web trust model support, feature location and requirements traceability respectively

Approaches	**External Info.**	**Soft. Repo.**	**CTW**	**SCP**	**BCRs**	**Mul. Exp.**	**AEW**	**Tool Supp.**	**ET**	**WTM Supp.**	**FL**	**RT**
Gethers *et al.* (2011)	✗	✗	✗	✗	✗	✓	✓	✗	✗	✗	✓	✗
De Lucia *et al.* (2011)	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓
McMillan *et al.* (2009)	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✓
Maletic and Collard (2009)	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓
Mader *et al.* (2008)	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✓
Poshyvanyk *et al.* (2007)	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✓	✗
Kagdi *et al.* (2007)	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓
Sherba *et al.* (2003)	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✓
Antoniol *et al.* (2002b)	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓
Sharif and Kagdi (2011)	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗
Uwano *et al.* (2006)	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗
Palmer *et al.* (2000)	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗
Berg and Van (2001)	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗

Part II

Creation of Experts and Combining their Opinions

CHAPTER 4

CREATION OF EXPERTS

This chapter explains how we can create experts using different sources of information. We provide the details of the four approaches, *i.e.*, Histrace, BCRTrace, Partrace, and developer's knowledge, to create experts. Each expert is capable of voting on the links recovered by an IR technique.

4.1 Histrace

Histrace creates links between the set of requirements, R and the source code, C, using the software repositories, *i.e.*, CVS/SVN commit messages and bug reports. Histrace considers the requirements' textual descriptions, CVS/SVN commit messages, bug reports, and classes as separate documents. It uses these sources of information to produce two experts $Histrace_{commits}$, and $Histrace_{bugs}$, which use CVS/SVN commit messages and bug reports, respectively, to create traceability links between R and C through the software repositories. Below, we discuss each step of Histrace in details.

CVS/SVN Commit Messages:

We use Ibdoos[1] to convert CVS/SVN commit logs into a unified format and put all commit messages into a database for ease of treatment. Ibdoos provides parsers for various formats of commit logs, including CVS, Git, and SVN.

To build $Histrace_{commits}$ expert, Histrace first extracts CVS/SVN commit logs and excludes those that (1) are tagged as "delete" because they concern classes that have been deleted from the system and thus cannot take part in any traceability link, (2) do not concern source code (*e.g.*, if a commit only contains HTML or PNG files), (3) have messages of length shorter or equal to one English word, because such short messages would not have enough semantics to participate in creating $Histrace_{commits}$.

Histrace then extracts the CVS/SVN commit messages as well as the classes' names that (1) are still part of the source code and (2) have been part of the commits. Histrace applies the same normalisation steps on CVS/SVN commit messages as those for requirements and source code (see Section ref subsection: pre-processing).

Histrace then uses an IR technique to link requirements to CVS/SVN log messages. For example, Histrace could use an IR technique, *e.g.*, VSM, to link the CVS/SVN log $Logs_{1741}$ of

1. http://www.ptidej.net/download/ibdoos/

Pooka, containing `NewMessageInfo.java`, to the requirement r_{21}. This means that Histrace can simply link r_{21} to `NewMessageInfo.java` and add it to the $Histrace_{commits}$ set for *Pooka*.

Bug Reports:

To build $Histrace_{bugs}$, Histrace extracts all the bug reports from a bug-tracking system. Usually, bug reports do not contain explicit information about the source code files that developers updated to fix a bug. All the details about the updated, deleted, or added source code files are stored in some CVS/SVN repository. Therefore, Histrace must link CVS/SVN commit messages to bug reports before being able to exploit bug reports for traceability.

Histrace uses regular expressions, *i.e.*, a simple text matching approach but with reasonable results, to link CVS/SVN commit messages to bug reports. However, Histrace could also use more complex techniques, such as those in (Bachmann *et al.* (2010); Wu *et al.* (2011)). Consequently, Histrace assumes that developers assigned to each bug a unique ID that is a sequence of digits recognisable via regular expressions. The same ID must be referred to by developers in the CVS/SVN commit messages.

Then, to link CVS/SVN commit messages to bug reports concretely, Histrace performs the following steps: (1) extracts all CVS/SVN commit messages, along with commit status and committed files, (2) extracts all the bug reports, along with time/date and textual descriptions, and (3) links each CVS/SVN commit message and bug reports using regular expressions, *e.g.*,

$$((b)[ug]\{0,2\}\backslash s*[id]\{0,3\}|id|fix|pr|\#)$$
$$[\backslash s\# =]*[?([0--9]\{4,6\})]?$$

which is the regular expression tuned to the naming and numbering conventions used by the developers of *Rhino*. This expression must be updated to match other naming and numbering conventions.

In its last step, Histrace removes false-positive links by imposing the following constraint:

$$fix(e[ds])?|bugs?|problems?|defects?patch''$$

This regular-expression constraint only keeps a CVS/SVN commit if it contains a keyword, *i.e.*, fix(es), fixed, bug(s), problem(s), defect(s), or patch, followed by a number, *i.e.*, if it follows naming conventions for bug numbering usual in open-source development. It thus returns the bug reports linked to CVS/SVN commit messages.

Histrace then uses an IR technique to link requirements to bug reports. For example, Histrace could use an IR technique to link the bug report bug_{434} to requirement r_{11}. Then, it could link the bug report bug_{434} to the CVS/SVN log $Logs_{4912}$ in *SIP* using the appropriate regular expression. $Logs_{4912}$ contains `FirstWizardPage.java` and `DictAccountRegistrationWizard.java`. Thus, Histrace could link r_{11} to `FirstWizardPage.java` and `DictAccountRegistrationWizard.java` to build the $Histrace_{bugs}$ set for *SIP*.

4.2 BCRTrace

The BCRTrace provides the binary class relationship (BCRs) between classes. Each BCR is treated as an expert that votes on the links recovered by an IR technique. Below we provide the definition of some notations: Let $B = \{b_1, \ldots, b_N\}$ be a set of high-level documents, *e.g.*, bug reports, $C = \{c_1, \ldots, c_M\}$ be a set of classes, and $R = \{r_1, \ldots, r_P\}$ be a set of BCRs.

Let $\mathcal{L} = \{L_1, \ldots, L_N\}$ be a set where each L_i is a set of classes $\{c_1, \ldots, c_j\}$ linked to a high-level document b_i; whose cosine similarity is greater than 0. Let $Q = \{Q_{1,1}, \ldots, Q_{N,P}\}$ be a set in which each $Q_{i,k}$ is a subset of L_i, in which all classes are linked by r_k. Q is produced by the BCRTrace.

The BCRTrace takes as input a baseline set, $\mathcal{L}$, provided by an IR technique and the source code or binary code of the program. It produces as output the set Q using analyses based on models of the source code (Guéhéneuc and Albin-Amiot (2004)). Below we explain the procedure to create BCRs among classes.

Step 1: PADL Model Creation (PADL Model Creator):

We use the Ptidej tool suite (Guéhéneuc (2005)) and its PADL meta-model (Patterns and Abstract-level Description Language) to build PADL models of OO programs. A PADL model is a representation of a program compliant with the PADL meta-model. Such model includes the main constituents that represent the structure and part of the behaviour of a program, *i.e.*,ie classes, interfaces, member classes and interfaces, methods, attributes, inheritance.

The Ptidej tool takes as input the C++ or Java source code or binaries of programs and generates PADL models of these programs. The Ptidej tool suite essentially divides into a set of parsers, an implementation of the PADL meta-model and language-dependent extensions to the meta-model to integrate, within a PADL model, constituents particular to a programming language. For example, the PADL meta-model does not define a constituent to describe C++ destructors, this constituent is provided along with the C++ parser to allow the modelling of C++ programs with possible highest precision. The C++ and Java source code parsers are based on the parsers provided by the CDT and JDT Eclipse plug-ins. The Java binary code parser uses the BCEL library.

Step 2: BCRs among classes recovering (BCRs Recovery):

Using PADL and based on an extensive literature review (Guéhéneuc and Albin-Amiot (2004)), we implemented analyses to uncover BCRs in the source code of programs and make them explicit in their PADL models.

Theoretically, we assume that a BCR exists between two classes if any method of one of the two classes invokes at least one method of the other. Then, we define four properties of any potential BCR but inheritance: we exclude inheritance because it is explicit in the source code of programs in C++ (through the `:` syntax) and Java (through the `extends` keyword). We need dedicated analyses to recover all BCRs but inheritance because, in mainstream programming languages, such as C++ and Java, these relationships are not explicit in the source code but implemented by developers from the design documents using various idioms.

These properties are: (1) exclusivity of the participation of the instances of the classes involved in the BCR, (2) the types of the receivers of the messages exchanged, (3) the life-time of the instances of the classes involved in the BCR, and (4) the multiplicity of the instances of the classes involved in the BCR. We use these properties to define uniquely each BCR, from the least constraining in terms of the values of the properties to the most constraining: use, association, aggregation, and composition.

We do not recall here the sets of values for each properties because the reader may find all details in a previous work (Guéhéneuc and Albin-Amiot (2004)). Also, we do not further consider composition because it requires dynamic information that would either be gathered through dynamic analyses or through incomplete static analyses. These properties and their values essentially allow identifying the various idioms used by developers to implement BCRs.

When defining the properties of any BCR (but inheritance and composition), we make sure that we can identify these values of the properties using PADL constituents, in particular: classes, methods, and fields, and method invocations between methods. Thus, our analyses mainly consist in identifying potential BCR among classes and then refining these candidates using the values of their properties. These analyses are source code analyses, because they use essentially information extracted from the source code. However, we abstract these analyses to make them operate on PADL models so that we can recover BCRs from various programming languages.

When applying these analyses on a PADL model, we obtain a new PADL model that contains all the constituents from the original model plus constituents representing explicitly the BCRs: instances of the Use, Association, Aggregation, and Implementation constituents of an extension to the PADL meta-model.

Step 3: Linked Classes Extraction (BCRs Filter):

Based on the model built in Step 1 and refined in Step 2 and the set $\mathcal{L}$ provided by an IR technique, we build the set Q. For each BCR r_k of R, we iterate over each L_i of $\mathcal{L}$. If the

PADL model of the program indicates for a class c_j of L_i a BCR r_k between c_j and another class c_l of L_i, then c_j and c_l are selected as elements of $Q_{i,k}$. At the end, of this step, we associate to each bug b_i four sets:

– $Q_{i,1}$, the classes linked by an inheritance relationship and linked to a high-level document b_i;
– $Q_{i,2}$, the classes linked by a use relationship and linked to a high-level document b_i;
– $Q_{i,3}$, the classes linked by an association and linked to a high-level document b_i;
– and, $Q_{i,4}$, the classes linked by an aggregation and linked to a high-level document b_i.

4.3 Partrace

Partrace is useful when there are no software repositories and BCRs available for a software system. Partrace uses the source code entities to create experts. It divides source code in multiple partitions and each partition contains chunk of source code identifiers.

Let $R = \{r_1, \ldots, r_n\}$ be a set of requirements, $\mathcal{C} = \{C_1, \ldots, C_m\}$ be a set of implementing classes. Following Bunge ontology Bunge (1977), let $X = \langle x, P(x)\rangle$ be a substantial individual *i.e.*, an object, where the object X is identified by its unique identifier x, and $P(x)$ a set of properties, in this dissertation, the collection of all source code partitions or collection combination thereof *i.e.*, all possible information sources. To define information sources, let ψ_i be a family of functions $i = 1, \ldots N$ each function selects a sub-set of X properties, for example, the class names and/or method names. In other words, each ψ_i function creates a new set of documents having some of the $P(X)$ properties. A developer could define the granularity of a partition, *i.e.*, number of ψ_i functions. For example, source code could be divided at the line level, so if there are 20 LOC then Partrace would create 20 experts.

To process source code, a Java parser is used to extract all source-code identifiers. The Java parser build an abstract syntax tree (AST) of the source code that can be queried to extract required identifiers, *e.g.*, class, method names, etc. In this dissertation we use Java source code and partitioned each file in four parts, *i.e.*, $\psi_i = i = 1, \ldots 4$, and textual information is stored in four separate files. Table 4.1 shows the source code partitions we use in this dissertation. An IR technique then creates links between a requirement, Java class file, and Partrace partitioned files. A link between a requirement and Java class file is reevaluated by the Partrace partitioned files' links.

Table 4.1 Source Code Sections used in Experimentation

Acronym	**Identifier Type**
CN	Class Name - one name per file
MN	All Public and Private Method Names of a Class
VN	Class and Method Variable Names of a Class
CMT	All Block and Single line Comments of a Class

4.4 Developer's Knowledge

IR-based RTAs are not 100% accurate yet. A developer has to manually verify the false positive links and recover the missing links. The developer has her own preferences and process (Wang *et al.* (2011)) to perform RT tasks and it could be different from an automated technique. For example, an IR technique would weight a term based on its frequency in the document, whereas, a developer may weight a term based on its location in the source code. Thus, integrating developer's knowledge in automated RTAs could yield better accuracy. In this dissertation, we use developer's knowledge as extra source of information to trust a link recovered by an IR technique.

There are many ways in which we could observe developers during RT tasks. For example, we could use Mylyn[2] log to analyse developer's activities, capture her monitor screen and record their voices (Wang *et al.* (2011)), or using eye-tracker (Sharif and Kagdi (2011)). In this dissertation, we use eye-tracker to observe developers during RT task. An eye-tracker provides us with two main types of eyes-related data: fixations and saccades. A fixation is the stabilisation of the eye on an object of interest for a period of time, whereas saccades are quick movements from one fixation to another. We can define area of interest on the screen and eye-tracker would monitor the total number of fixations a developer has on each area of interest. If a developer has more fixation points then we could assume that it is important for her. After analysing the important parts of source code for a developer, we could utilise this information in automated IR-based RTAs. For example, we could define new improved weighting schemes based on eye-tracking results and we could define a process that where to look first on the source code during RT task.

In this dissertation, we use FaceLab. Facelab from Seeing Machine[3] is a video-based remote eye-tracking. It consists of two built-in cameras, one infrared pad, and one computer. Facelab tracks developer's eye-movements by capturing developers' head using facial features, including nose, eye-brows, and lips. Facelab transmits eye-movements data to a data visu-

2. http://www.eclipse.org/mylyn/
3. http://www.seeingmachines.com/product/facelab/

alisation tool, gaze tracker from eye response. Gazetracker stores the fixations and saccades associated with each image and display all fixations in the foreground. We use two 24" LCD monitors: the first one is used by the experimenter to set up and run the experiments while monitoring the quality of the eye-tracking data. The second monitor (screen resolution is 1920 x 1080) for displaying the RT task. The stimulus display was maximised to cover the entire screen.

4.5 Summary

In this chapter, we showed that we can create experts using different sources of information. Each expert is independent and could be used in a different scenario. For example, if a software system does not have repository then BCRTrace or Partrace could be used to create experts. All these experts could vote on the baseline links created by an IR technique.

CHAPTER 5

COMBINING EXPERTS' OPINIONS

In previous chapter, we explained the procedure to create experts using each source of information. Each expert is a separate entity and can vote on a baseline link created by IR technique. However, to reevaluate the trustworthiness of a baseline link, we must combine the opinions of all the experts. In this chapter, we propose a trust model, *i.e.*, Trumo. Trumo is a technique to re-rank the traceability links recovered by an IR technique between requirements and source code using a trust model. Trumo is inspired by a users' Web trust models (Berg and Van (2001); McKnight *et al.* (2002); Palmer *et al.* (2000); Koufaris and Hampton-Sosa (2004)). Trumo considers experts to give their opinions on the baseline traceability links recovered using the IR technique. Following we present the basic definition of mathematical symbols that we used in our trust model and Trumo model.

5.1 Definitions

We represent a traceability link as a triple {source document, target document, similarity} and we use the following notations. Let $R = \{r_1, \ldots, r_N\}$ be a set of requirements and $C = \{c_1, \ldots, c_M\}$ be a set of classes supposed to implement these requirements. Let $\mathcal{T} = \{T_1, \ldots, T_P\}$ be a collection of sets where each $T_i = \{t_1, \ldots, t_{N_i}\}$ is a set of homogeneous pieces of information, *e.g.*, the set of all bug reports or of BCRs for a given system. P is the total number of experts.

Then, let us assume that, for each $T_i \in \mathcal{T}$, we define a function δ_{T_i} mapping one element of T_i into a subset of C. For example, if T_i is a set of bug reports, then, for a given bug report t_k, $\delta_{T_i}(t_k)$ returns the set of classes affected by t_k, with $\delta_{T_i}(t_k) \subseteq C$.

Let $R2C$ be the set of baseline traceability links recovered between R and C by an standard IR technique, such as VSM, and, further, assume that, for each set $T_i \in \mathcal{T}$, we build a set $R2CT_{i,r_j,t_k}$ for each expert T_i as follows:

$$R2CT_{i,r_j,t_k} = \{(r_j, c_s, \sigma'_i(r_j, t_k)) | c_s \in \delta_{T_i}(t_k) \ \& \ t_k \in T_i\}$$

Finally, let us define two functions α and ϕ. The first function, $\alpha(r_j, c_s, \sigma'(r_j, c_s))$, returns the pair of documents (or set of pairs) involved in a link, *e.g.*, requirements and source code, *i.e.*, (r_j, c_s). The second function, $\phi(r_j, c_s, \sigma'(r_j, c_s))$, returns the similarity score $\sigma'(r_j, c_s)$ of the link.

Given these definitions, Trustrace works as follows. It first builds the set $R2C$ from R and C using an IR technique. Then, it calls an expert, *e.g.*, Histrace, to build the sets $R2CT_{i,r_j,t_k}$. Second, it uses a trust model, *i.e.*, Trumo, to evaluate the trustworthiness of each link using a weighting scheme, *i.e.*, DynWing, to compute the weights $\lambda_i(r_j, c_s)$ to assign to each link in the sets $R2CT_{i,r_j,t_k}$, and to re-rank the similarity values of the link, using the experts' opinions $R2CT_{i,r_j,t_k}$.

5.2 Trumo

Trumo assumes that different experts, *e.g.*, $Histrace_{commits}$ (also known as $R2CT_{1,r_j,t_k}$) and $Histrace_{bugs}$ ($R2CT_{2,r_j,t_k}$), know useful information to discard or re-rank the traceability links between two documents, *e.g.*, requirements and source code classes. Trumo is thus similar to a Web model of the users' trust: the more users buy from a Web merchant, the higher the users' trust of this merchant (Koufaris and Hampton-Sosa (2004)).

By the definitions in Section 5.1, in Equation 5.1, r_j is a requirement with $r_j \in R$; c_s is a class with $c_s \in \delta_{T_i}(t_k)$ because we use the sets $T_i \in \mathcal{T}$ to build a set of trustable links Tr; σ'_i is the similarity score between the requirement r_j and some class t_k such that $t_k \in T_i$ and $\alpha(R2CT_{i,r_j,t_k})$ returns a pair (r_j, t_k). With Equation 5.1, Trumo uses the set of candidate links $l_{rc} = (r_j, c_s, \sigma_i(r_j, c_s))$ with $j \in [1, \ldots, N]$ and $s \in [1, \ldots, M]$ and the sets of candidate links $l_{rt} = (r_j, c_s, \sigma'_i(r_j, t_k))$ with $j \in [1, \ldots, N]$ and $k \in [1, \ldots, N_i]$ generated from each expert T_i and for each requirement $r_j \in R$.

$$
\begin{aligned}
Tr = \{(r_j, c_s, \sigma'_i(r_j, t_k)) \quad & | \\
\exists\ t_k \in T_i : (r_j, c_s) \quad & \in \quad \alpha(R2CT_{i,r_j,t_k}) \\
& \& \quad (r_j, c_s) \in \alpha(R2C)\}
\end{aligned}
\tag{5.1}
$$

Figure 5.1 shows the Venn diagram of the $R2C$, $R2CT_{i,r_j,t_k}$, and Tr sets. The last constraint $(r_j, c_s) \in \alpha(R2C)$ imposes that a link be present in the baseline set $R2C$ and in any of the $R2CT_{i,r_j,t_k}$ sets. If a link does not satisfy this constraint, then Trumo discards it. Then, Trumo re-ranks the similarity of the remaining links in Tr as follows. Let $TC_i(r_j, c_s)$ be the restriction of Tr on (r_j, c_s) for the source T_i, *i.e.*, the set $\{(r_j, c_s, \sigma'_i(r_j, t_k)) \in Tr\}$, then Trumo assigns to the links in $TC_i(r_j, c_s)$ a new similarity $\sigma^*_i(r_j, c_s)$ computed as:

$$
\sigma^*_i(r_j, c_s) = \frac{\sigma(r_j, c_s) + \sum_{l \in TC_i(r_j, c_s)} \phi(l)}{1 + |TC_i(r_j, c_s)|}
\tag{5.2}
$$

where $\sigma(r_j, c_s)$ is the similarity between the requirement r_j and the class c_s as computed

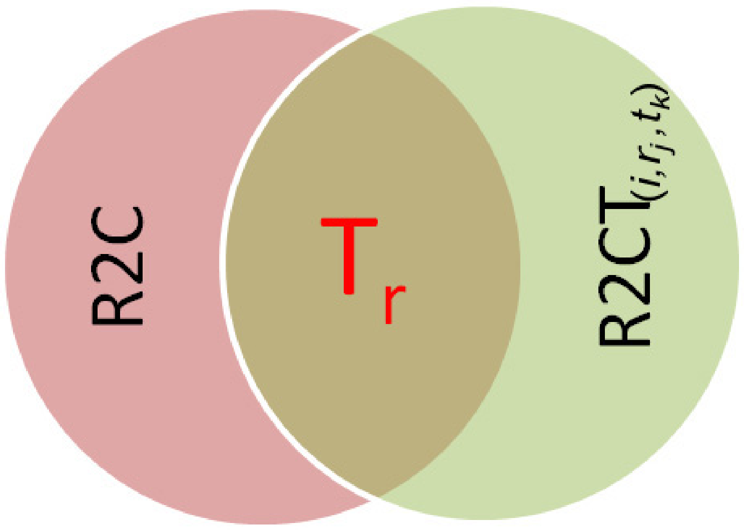


Figure 5.1 Overlapping of $R2C$, $R2CT_{i,r_j,t_k}$, and Tr

in $R2C$ and $\phi(l)$ is the similarity of the documents linked by the link l of $TC_i(r_j, c_s)$, *i.e.*, derived from t_k, which means $\sigma_i'(r_j, t_k)$. Finally, $|TC_i(r_j, c_s)|$ is the number of elements in $TC_i(r_j, c_s)$. The higher the evidence (*i.e.*, $\sum_{l \in TC_i(r_j,c_s)} \phi(l)$) provided by links in $TC_i(r_j, c_s)$, the higher the new similarity $\sigma_i^*(r_j, c_s)$; in the contrary, little evidence decreases $\sigma_i^*(r_j, c_s)$.

Finally, Trumo assigns weight to each expert and combines their similarity values to assign a new similarity value to each link in Tr using the ψ function:

$$\begin{aligned}\psi_{r_j,c_s}(Tr) &= \Big[\sum_{i=1}^{P} \lambda_i(r_j, c_s)\sigma_i^*(r_j, c_s)\Big] \\ &+ \lambda_{P+1}(r_j, c_s)\frac{|Tr(r_j, c_s)|}{\max_{n,m} |Tr(r_n, c_m)|}\end{aligned} \tag{5.3}$$

where $Tr(r_j, c_s)$ is the subset of Tr restricted to the links between r_j and c_s; $\lambda_i(r_j, c_s) \in [0, 1]$ and $\lambda_1(r_j, c_s) + \lambda_2(r_j, c_s) + ... + \lambda_{P+1}(r_j, c_s) = 1$; recall that P is the total number of experts. With the ψ function, the more often a pair (r_j, c_s) exists in Tr, the more we can trust this link (if such a link is also present in $R2C$).

5.3 Voting

Voting is a non-weighted instance of Trumo. It does not require any kind of weighting to combine experts' opinion. Voting assumes that traceability links have already been recovered between elements of C and R by an IR technique; let this set be a $\mathcal{L} = \{l_1, \ldots, l_M\}$ set of all links where M can be as high as $n \times m$. Voting uses the expert(s) β_i to score each link l_q and decide via majority voting if the links is likely to be a real link or should be rejected.

Suppose that VSM creates a link between a requirement R_1 "adding prelim support for spam filters" and a class "SpamFilter". Further assume that Voting uses CN, MN, VN, and CMT (see Section 4.3) as experts. For Voting, links l_q are the base links, experts' vote and validate to confirm these links. To accept a link, at least two β_i must agree on a link l_q. Let us further assume, that CN and MN are the top two experts and that there are no ties, *i.e.*, this means Voting only uses CN and MN as experts. Then CN and MN will vote on the given link, this is to say the link between a requirement R_1 and a class `SpamFilter`, will be accepted if and only if both CN and MN experts also return a non-zero similarity between `SpamFilter` and R_1. If the links is accepted then it will be assigned the highest similarity among the three computed *i.e.*, standard VSM, CN, and MN.

5.4 Weighting Technique

Combining experts' opinion is still an open research question. Several researchers (Poshyvanyk *et al.* (2007); Gethers *et al.* (2011)) proposed static weighting and principal component analysis based weighting techniques this problem. However, the accuracy of these techniques is still not up to the mark. We also try to solve this problem by converting this problem as maximisation problem to combine experts and propose a novel weighting technique, *i.e.*, DynWing. Below we explain each weighting technique in detail.

5.4.1 Static Weight

In static weighting techniques (Ali *et al.* (2011b); Poshyvanyk *et al.* (2007)) a software engineer can define a static weight, *e.g.*, 0.3 or 0.7, to an expert. However, to assign a best weighting combination to expert requires an oracle. Because, if only with the help of oracle a software engineer would know the current combination does not provide better accuracy. The starting weighting combination depends on software engineer's guts feeling. For example, if there are two experts, *e.g.*, Partrace and Histrace, a software engineer can assign 0.4 to Partrace and 0.6 to Histrace. Then, the software engineer use oracle to verify if this combination produce better accuracy else he needs to try another weight combination.

(Ali *et al.* (2011b); Poshyvanyk *et al.* (2007)) used static weighting techniques and provided a range of weight. The authors mentioned that this range could be used to assign weights to different experts. However, that range could be different for different experts and–or datasets. Thus, we cannot generalise the weighting range. In addition, if there are two experts then a software engineer can try different combinations to assign weights to experts. But as the number of experts grows then it becomes difficult to try all the combination even with the help of an oracle.

5.4.2 DynWing

To automatically decide the weights $\lambda_i(r_j, c_s)$ for each expert, we apply a dynamic weighting technique. Existing techniques (Ali *et al.* (2011b); Poshyvanyk *et al.* (2007)) to define weights use static weights for all the experts. Thus, they require oracles to decide a "good" weight or range of weights. However, with real, legacy systems, no such oracle exists, *i.e.*, no a priori-known set of traceability links exists. Moreover, using the same static weight may not be beneficial for all the recovered links.

Therefore, we consider each link recovered by a baseline IR technique and by the different experts as an independent link and dynamically assign weights to baseline links and each expert. Choosing the right weight per link is a problem that we formulate as a maximisation problem. Basically, we have different experts, *i.e.*, CVS/SVN commits , bug reports, and others, to trust a link. Each expert has its own trust into the link. By maximising the similarity value $\psi_{r_j,c_s}(Tr)$ (and hence determining the optimal $\lambda_i(r_j, c_s)$ values), DynWing automatically identifies the experts that are most trustworthy (highest $\lambda_i(r_j, c_s)$ values) and those that are less trustworthy (lowest $\lambda_i(r_j, c_s)$ values):

$$\max_{\lambda_1(r_j,c_s),\ldots,\lambda_{P+1}(r_j,c_s)} \{\psi_{r_j,c_s}(Tr)\} \tag{5.4}$$

with the following constraints:

$$\begin{aligned} 0 \leq \lambda_i(r_j, c_s) \leq 1, i = 1, ..., P+1 \\ \lambda_1(r_j, c_s) + \lambda_2(r_j, c_s) + ... + \lambda_{P+1}(r_j, c_s) = 1 \\ \lambda_{k_1}(r_j, c_s) \geq \lambda_{k_2}(r_j, c_s) \geq \ldots \geq \lambda_{k_{P+1}}(r_j, c_s) \end{aligned}$$

Given the three previous constraints, it is possible that DynWing assigns $\lambda_i(r_j, c_s) = 1$ to a single expert i. To avoid such an assignment, a developer can define her global trust into the experts. For example, CVS/SVN commit messages may be considered by the developer more trustworthy than bug reports. Therefore, the developer may constrain further Equation 5.4 by imposing:

$$\lambda_{commits}(r_j, c_s) \geq \lambda_{bugs}(r_j, c_s) > 0$$

Table 5.1 Different combinations of experts, Trumo, and weighting techniques

Name of Combination	Expert	Trust-Model			Weighting Techniques		
		Trumo	Trumo$_{Ranker}$	Voting	DynWing	PCA	Static
Trustrace	Histrace	✓			✓	✓	✓
LIBCROOS	BCRTrace		✓				✓
COPARVO	Partrace			✓			

5.4.3 Principal Component Analysis

Principal-component analysis (PCA) is a mathematical analysis used in a recent case study on traceability (Gethers *et al.* (2011)) to combine different IR techniques and to define weights for each technique. PCA defines a single static weight for each expert.

PCA uses an orthogonal transformation to convert a set of correlated variables into a set of values of un-correlated variables, called principal components. This transformation is defined in such a way that the proportion of variance for each principal component (PC) is $PC_1 > PC_2 > \dots PC_n$. The number of PCs is less than or equal to the number of original variables.

Gethers *et al.* (Gethers *et al.* (2011)) compute the weights for different IR techniques using PCA as follows: (1) they use the value of the proportion of variance based on the PC with the highest correlation and (2) they normalise values to obtain weights for each technique. For example, if PC_1's proportion of variance is 71.29% and CVS/SVN has a correlation of 0.99 with PC_1, we would assign 71.29% to CVS/SVN. Likewise, if a bug report has a higher correlation than PC_1 when compared to other PCs, it would also receive a value of 71.29%. After assigning values of proportions of variances to all experts, we normalise the values so their sum equals to one.

5.5 Combination of Experts, Trumo, and Weighting Techniques

In this dissertation, we try different experts with different instances of Trumo model and weighting techniques. Table 5.1 shows the different combinations and name of each combination. It is quite difficult to try all the combinations, thus we made all the combinations simple and unique to evaluate all the combinations. For example, we use Partrace expert with voting and call it COPARVO.

Part III

Usage of Experts and their Opinions

CHAPTER 6

ASSESSING TRUSTRACE AS A TRACEABILITY RECOVERY METHOD

During software maintenance and evolution, developers often evolve requirements and source code differently. Indeed, they often do not update requirements and requirement-traceability links with source code. Yet, while developers may not evolve requirements in synchronisation with source code, they frequently update other sources of information, including CVS/SVN repositories, bug-tracking systems, mailing lists, forums, and blogs. We believe that we can exploit these other sources of information to build improved traceability-recovery approaches.

Consequently, we conjecture that: (1) we can mine software repositories, *e.g.*, CVS/SVN repositories, to support the traceability recovery process and improve the precision and recall of IR-based traceability recovery approaches; (2) we can think of heterogeneous sources of information as experts whose opinions we can combine using a trust model to discard/re-rank the traceability links provided by the IR techniques to improve accuracy; and, (3) we can use an automatic, dynamic, per-link weighting technique rather than some global static weights to combine the opinions of experts to avoid the need of manually-built oracles to tune weights.

Consequently, we design, implement, and evaluate Trustrace, a traceability-recovery approach between requirements and source code, which we use to support our conjectures. Trustrace uses heterogeneous sources of information to dynamically discard/re-rank the traceability links reported by an IR technique. Trustrace consists of Histrace, Trumo, and DynWing.

We empirically evaluate Trustrace on four medium-size open-source systems *i.e.*, *jEdit v4.3*, *Pooka v2.0*, *Rhino v1.6*, and *SIP Communicator v1.0-draft*, to compare the accuracy of its recovered requirement traceability links with those of state-of-the-art IR techniques. As state-of-the-art IR techniques, we choose the VSM, a representative of the algebraic family of techniques, and JSM, a representative of the probabilistic family of techniques. We use the IR measures of precision, recall, and the F_1 score. We also compare two different weighting techniques: PCA and DynWing (see Section 5.4). We thus report evidence that Trustrace improves, with statistical significance, the precision and recall of the recovered traceability links.

Hence, we found the evidence supporting our three conjectures about the benefits of (1) mining software repositories and considering the links recovered through these repositories

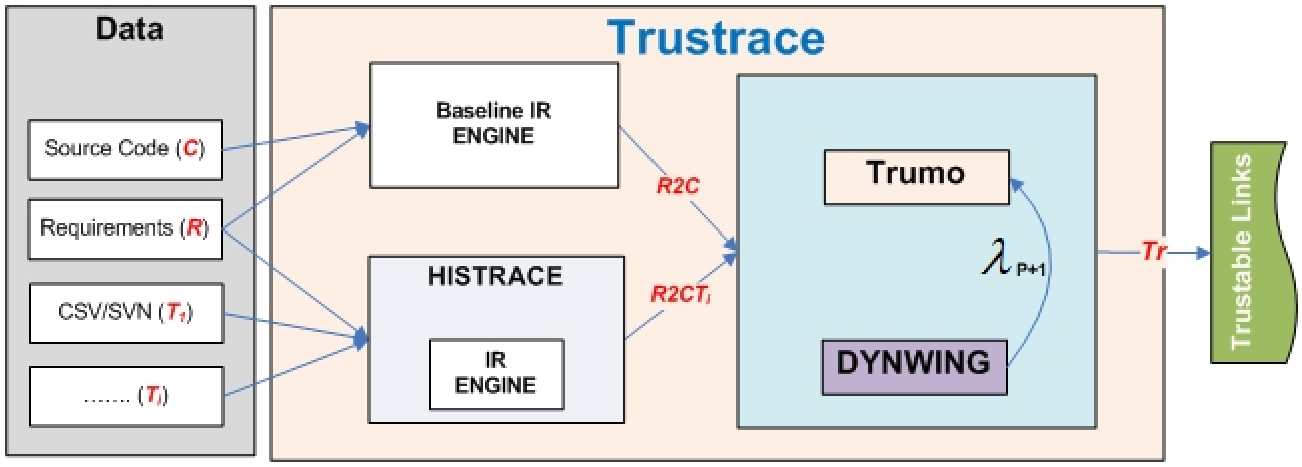


Figure 6.1 Trust-based requirement traceability process

as experts, (2) using a trust model inspired by Web-trust models to combine these experts' opinions, and (3) weighting the experts' opinions dynamically for each link recovered using an IR technique.

The rest of the chapter is organised as follows: Section 6.1 describes our approach, Trustrace, and usage of our two novel techniques: Histrace and Trumo. Section 6.2 presents our empirical evaluation of Trustrace while Section 6.3 reports its results. Section 6.4 provides a discussion of the results and a qualitative analysis. Finally, Section 6.5 provides the summary of the chapter.

6.1 Trustrace: Trust-based Traceability

We now present Trustrace. Trustrace uses software repositories, *e.g.*, CVS/SVN repositories and bug-tracking systems, as experts to trust more or less some baseline links recovered by an IR technique and, thus, to discard/re-rank the links to improve the precision and recall of the IR-based techniques. Figure 6.1 shows the high-level architecture of Trustrace, whose conceptual steps we detail in the following sections.

In the following, without the loss of generality, we target OO systems and use classes as representative of source code files. We also consider classes because considering packages is likely to be too coarse-grained, as a package contributes to the implementation of several requirements, while considering methods is likely to be too fine-grained as a method only participates in the implementation of some requirement(s), rarely implements them entirely. Moreover, CVS/SVN software repositories only consider files, not packages or methods.

Trustrace uses IR techniques for two different purposes: (1) to create the baseline set of traceability links $R2C$, whose similarity values will be recomputed using Trumo and DynWing using the output of Histrace and (2) to create the output sets of Histrace: $R2CT_{i,r_j,t_k}$. Trustrace does not dependent on a specific IR technique. Different IR techniques (Antoniol *et al.* (2002b); Abadi *et al.* (2008); Marcus and Maletic (2003c); Asuncion *et al.* (2010)) can be used to create the traceability links between two documents.

6.1.1 Usage of Histrace

Histrace creates experts using software repositories T_i (see Section 4.1), *i.e.*, in the following, T_1 stands for CVS/SVN commit messages and T_2 for bug reports. It creates two experts $R2CT_{1,r_j,t_k}$, which we call $Histrace_{commits}$ in the following for simplicity, and $R2CT_{2,r_j,t_k}$, which we call $Histrace_{bugs}$.

Depending on the input information source (*i.e.*, requirements, source code, CVS/SVN commit messages, or bug reports), we perform specific pre-processing steps to remove irrelevant details from the source, (*e.g.*, CVS/SVN commit number, source code punctuation or language keywords), split identifiers, and, finally, normalise the resulting text using stop-word removal and stemming. In particular, we perform the pre-processing steps for requirements and source code as described in Section 2.1.1.

CVS/SVN Commit Messages: Following our definitions, T_1 is the set of all filtered CVS/SVN commit messages (see Section 4.1) and, for any $t_k \in T_1$, we have a function δ_{T_1} that returns a subset of the classes $C'_k \subset C = \{c_1, \ldots, c_M\}$ modified in the commits. Histrace uses commit messages and an IR-based technique to compute σ'_{r_i,t_k}. Using δ_{T_1}, Histrace builds $R2CT_{1,r_j,t_k}$.

Bug Reports: Following our definitions, T_2 is the set of all bug reports (see Section 4.1) and, for any $t_k \in T_2$, we have a function δ_{T_2} that returns a subset of classes $C'_k \subset C = \{c_1, \ldots, c_M\}$ modified to fix a bug. Histrace uses the bug reports and an IR-based technique to compute σ'_{r_i,t_k}. Using δ_{T_1}, Histrace builds $R2CT_{2,r_j,t_k}$ as explained in 4.1.

6.2 Empirical Evaluation

We now report on an empirical evaluation of Trustrace with four systems to assess its accuracy in terms of precision and recall. We use two state-of-the-art IR techniques, *i.e.*, JSM and VSM, for evaluation purposes. We use the names $Trustrace_{VSM}$ and $Trustrace_{JSM}$ to denote the IR techniques that Trustrace uses. We also compare the DynWing weighting technique of Trustrace with the principal component analysis (PCA) weighting technique (Gethers *et al.* (2011)).

We implement Trustrace and its three novel techniques in FacTrace (see Section 1.5).

6.2.1 Goal

The *goal* of our empirical evaluation is to study the accuracy of Trustrace when recovering traceability links against that of a single IR technique, JSM and VSM, using requirements, source code, CVS/SVN commits, and–or bug reports as experts. The *quality focus* is the accuracy of Trustrace in terms of precision and recall (Frakes and Baeza-Yates (1992)). It is also the improvement brought by the dynamic weighting technique, DynWing, with respect to a PCA-based technique in terms of F_1 score. The *perspective* is that of practitioners interested in recovering traceability links with greater precision and recall values than that of currently-available traceability recovery IR-based techniques. It is also that of researchers interested in understanding whether or not we can support our conjectures.

6.2.2 Research Questions

Our research questions are:

– **RQ1:** How does the accuracy of the traceability links recovered by Trustrace compare with that of approaches based on JSM and VSM alone?

– **RQ2:** How does the accuracy of the traceability links recovered using DynWing compare to that using PCA?

To answer **RQ1**, we assess the accuracy of JSM, Trustrace, and VSM in terms of precision and recall by applying them on four systems seeking to reject the four null hypotheses:

– $\mathbf{H_{01}}$**:** There is no difference in the precision of the recovered traceability links when using Trustrace or VSM.

– $\mathbf{H_{02}}$**:** There is no difference in the precision of the recovered traceability links when using Trustrace or JSM.

– $\mathbf{H_{03}}$**:** There is no difference in the recall of the recovered traceability links when using Trustrace or VSM.

– $\mathbf{H_{04}}$**:** There is no difference in the recall of the recovered traceability links when using Trustrace or JSM.

To answer **RQ2**, we use the DynWing and PCA weighting techniques and compute the F_1 score of Trustrace to analyse which weighting technique provides better results. We try to reject the two null hypotheses:

– $\mathbf{H_{05}}$**:** DynWing does not provide automatically better $\lambda_i(r_j, c_s)$ than a PCA-based weighting technique for $Trustrace_{VSM}$.

– $\mathbf{H_{06}}$**:** DynWing does not provide automatically better $\lambda_i(r_j, c_s)$ than a PCA-based weighting technique for $Trustrace_{JSM}$.

Variables:

We use precision, recall, and F_1 (see Section 2.4.1 and 2.4.2) score as dependent variables. All measures have values in $]0, 1]$: We use the F_1 score to compare DynWing and PCA because F_1 equally weighs precision and recall. Thus, it shows which weighting technique provides both best precision and recall values.

We use the approaches, either single IR techniques, *i.e.*, JSM and VSM, or Trustrace, as independent variables. The independent variable corresponding to Trustrace also includes varying values of λ_i using DynWing and PCA-based weighting techniques.

Objects:

We select the four open-source systems, *jEdit 4.3*, *Pooka v2.0*, *Rhino 1.6*, and *SIP Communicator 1.0-draft*, because they satisfy our key criteria. First, we choose open-source systems, so that other researchers can replicate our evaluation. Second, all systems are small enough so that we could recover and validate their traceability links manually in previous work (Ali *et al.* (2011b)), while still being a real-world system. The details about all the four datasets are available in Appendix A.

Oracles:

We use four oracles, $\text{Oracle}_{\text{jEdit}}$, $\text{Oracle}_{\text{Pooka}}$, $\text{Oracle}_{\text{Rhino}}$, and $\text{Oracle}_{\text{SIP}}$, to compute the precision and recall values of JSM, Trustrace, and VSM when applied on our four object systems. The details about the creation of oracles are explained in Appendix A.

6.2.3 Pre-Processing

We now detail how we gather and prepare the input data necessary to perform our empirical evaluation of Trustrace.

Requirements

jEdit contains 34 requirements. These requirements were manually identified and extracted from the *jEdit* source code repository (Dit *et al.* (2011b)). In previous work (Ali *et al.* (2011b)), we used PREREQUIR (Hayes *et al.* (2008)) to recover requirements for *Pooka* and *SIP*. We recovered 90 and 82 requirements for both systems, respectively. *Rhino* contains 268 requirements that we extracted from the related ECMAScript specifications by considering each ECMAScript section as a requirement.

Source Code

We downloaded the source code of *jEdit v4.3*, *Pooka v2.0*, *Rhino v1.6*, and *SIP v1.0-draft* from their respective CVS/SVN repositories. We made sure that we had the correct files for each system before building traceability links by setting up the appropriate environments and by downloading the appropriate libraries. We thus could compile and run all the systems.

CVS/SVN Commit Messages

Figure 6.2 shows an excerpt of a commit of *Pooka*. There are 3, 762, 1, 743, 3, 261, and 8, 079 SVN commits for *jEdit*, *Pooka*, *Rhino*, and *SIP*, respectively. We performed the data pre-processing steps described in Section 2.1.1 on all SVN commits with the help of FacTrace.

After performing the pre-processing steps, we obtained 2, 911, 1, 393, 2, 508, and 5, 188 SVN commits for *jEdit*, *Pooka*, *Rhino*, and *SIP*, respectively. There were many SVN commits that did not concern source code files. Also, some commit messages contained both source code files and other files. For example, revision 1604 in *Pooka* points only to HTML files except for one Java file, `FolderInternalFrame.java`. Therefore, we only kept the Java file and removed any reference to the HTML files. We stored all filtered SVN commit messages and related files in a FacTrace database.

Bug Reports

We cannot use *jEdit* (Dit *et al.* (2011b)) and *Pooka* bug reports because the first system does not have a publicly-available bug repository and the second one has too few recorded bugs (16). *Rhino* is part of the Mozilla browser and its bug reports are available via the Mozilla Bugzilla bug tracker. We extracted all the 770 bugs reported against *Rhino* and used Histrace to link them with the CVS repository as described in Section 6.1.1. Histrace automatically linked 457 of the bug reports to their respective commits. In the case of *SIP*, we downloaded 413 bug reports. *SIP* developers did not follow any rule while fixing bugs to link bug reports and commits. Hence, there was no bug ID in the commit messages. However, developers referenced SVN revision numbers in the bug reports' comments, *e.g.*, bug ID 237 contains the revision ID *r*4550. We tuned the regular expression of Histrace to find the revision IDs in the descriptions of the SIP bug reports. Histrace thus extracted all the bug IDs and linked them to SVN commits. Overall, Histrace automatically linked 169 bugs reported against *SIP* to their respective commits.

Last Pre-processing Step

We automatically extracted all the identifiers from the *jEdit*, *Pooka*, *Rhino*, and *SIP* requirements, source code, filtered CSV/SVN commit messages, and filtered bug reports, using FacTrace. The output of this step are four corpora that we use for creating traceability links as explained in Sections 6.1.1 and 5.2.

```
<logentry revision="1741">
    <author>akp</author>
    <date>2008-10-18T16:14:08.529138Z</date>
    <paths>
        <path kind="" action="M">/trunk/pooka/todo</path>
        <path kind="" action="M">/trunk/pooka/src/net/
            suberic/pooka/gui/NewMessageDisplayPanel.java
        </path>
        <path kind="" action="M">/trunk/pooka/src/net/
            suberic/pooka/NewMessageInfo.java
        </path>
    </paths>
    <msg>
        fixing a bug where the cc: on reply-all doesn't
        get cleared out if the cc field is set to empty.
    </msg>
</logentry>
```

Figure 6.2 Excerpt of *Pooka* SVN Log

6.2.4 Building Sets of Traceability Links

First, we use JSM and VSM to create traceability links, *i.e.*, $R2C_{JSM}$ and $R2C_{VSM}$, between requirements and source code. Second, we apply $Histrace_{commits}$, as described in Section 6.1.1, to process *jEdit*, *Pooka*, *Rhino*, and *SIP* CVS/SVN commit messages (T_1), and requirements to create the traceability link set $R2CT_{1,r_j,t_k}$. We process *SIP* and *Rhino* bug reports (T_2) to create the traceability link sets $R2CT_{2,r_j,t_k}$ using $Histrace_{bug}$.

For example, we trace *Pooka* requirement "*it should have spam filter option*" to the SVN commit message "*adding prelim support for spam filters*", SVN commit revision number 1133. Then, we recover all the source code classes related to this commit, *i.e.*, `SpamSearchTerm.java` and `SpamFilter.java`. Finally, we create a direct traceability link between the files `SpamSearchTerm.java` and `SpamFilter.java` to the requirement "*it should have spam filter option*".

Third, we apply Trumo as described in Section 5.2 using traceability links recovered with JSM and VSM. We thus compute two sets $R2C$: ($R2C_{JSM}$ and $R2C_{VSM}$), one with each IR technique. We then apply the Trumo equation via CVS/SVN commit messages and–or bug reports to discard/re-rank links by computing new similarity values using Equation 7.1. These values help to answer **RQ1** and to attempt rejecting our null hypotheses.

6.2.5 Experimental Settings

We must choose only one setting before applying Trustrace: our global trust of the experts. This global trust helps DynWing to assign weights to each link according to our a-priori trust in each expert as mentioned in Section 5.4.2. In our empirical evaluation, we define the global trust as:

$$\lambda_{commits} \geq \lambda_{bugs} \geq \lambda_{p+1} > 0$$

This global weight ensures that DynWing gives more weight to the $Histrace_{commits}$ expert than to the $Histrace_{bugs}$ expert and total number of times a link appears, *i.e.*, λ_{p+1}. We choose to favor $Histrace_{commits}$ based on the quality of the semantic information contained in the commit messages, the bug reports, and the source code classes, as further discussed in Section 6.4.1.

6.2.6 Analysis Method

We use $\text{Oracle}_{\text{jEdit}}$, $\text{Oracle}_{\text{Pooka}}$, $\text{Oracle}_{\text{Rhino}}$, and $\text{Oracle}_{\text{SIP}}$ to compute the precision and recall values of the links recovered using JSM, Trustrace, and VSM. JSM and VSM assign a similarity value to each and every traceability link whereas Trustrace uses its model, defined in Section 5.2, to reevaluate the similarity values of the links provided by a baseline technique.

To answer **RQ1**, we perform several experiments with different threshold (t) values, as explained in 2.3, on the recovered links to perform statistical tests on precision and recall values. Then, we assess whether the differences in precision and recall values, in function of t, are statistically significant between the JSM, Trustrace, and VSM approaches. To select an appropriate statistical test, we use the Shapiro-Wilk test to analyse the distributions of our data points. We observe that these distributions do not follow a normal distribution. Thus, we use a non-parametric test, *i.e.*, Mann-whitney test, to test our null hypotheses to answer **RQ1**.

An improvement might be statistically significant but it is also important to estimate the magnitude of the difference between the accuracy levels achieved with a single IR technique and Trustrace. We use a non-parametric effect size measure for ordinal data, *i.e.*, Cliff's d (De Lucia *et al.* (2011)), to compute the magnitude of the effect of Trustrace on precision and recall as follows:

$$d = \left| \frac{\#(x_1 > x_2) - \#(x_1 < x_2)}{n_1 n_2} \right|$$

where x_1 and x_2 are precision or recall values with JSM, Trustrace, and VSM, and n_1 and n_2 are the sizes of the sample groups. The effect size is considered small for $0.15 \leq d < 0.33$, medium for $0.33 \leq d < 0.47$ and large for $d \geq 0.47$.

To answer **RQ2**, we use PCA and DynWing to assign weights to the traceability links recovered using Trustrace. We use different values of t from 0.01 to 1 per steps of 0.01 to obtain different sets of traceability links with varying F_1 scores. We use the Mann-whitney to reject the null hypotheses $\mathbf{H_{05}}$ and $\mathbf{H_{06}}$.

6.3 Results

We now present the results and answers to our two research questions.

RQ1: How does the accuracy of the traceability links recovered by Trustrace compare with that of approaches based on JSM and VSM alone?

Figure 6.3 shows the precision and recall graphs of JSM, Trustrace, and VSM. Trustrace provides better precision and recall values than the two IR techniques by themselves. Table 6.1 shows the average precision and recall values calculated by comparing the differences between the JSM, Trustrace, and VSM approaches. Trustrace with DynWing has a better precision and recall than the other weighting techniques. The recall value for *Pooka* improves on average but without statistical significance when compared to VSM results.

In the case of *SIP* with only the $Histrace_{bugs}$ expert, recall values decrease with statistical significance when compared to VSM values, as discussed in Section 6.4.1. There is no statistically-significant decrease in recall when compared to JSM results. We explored the reason for the recall values to decrease in the case of *SIP* with $Histrace_{bugs}$. We found that only 4% of *SIP* SVN commits are linked to bug reports. Therefore, we did not find much evidence for many links and this lack of evidence yielded many links from the baseline set to be removed, following our constraints in Equation 5.1, and, consequently, a lower recall.

We performed the statistical tests described in Section 6.2.6 to verify whether or not the average improvements in precision and recall are statistical significant. We have statistically-significant evidence to reject $\mathbf{H_{01}}$ and $\mathbf{H_{02}}$. Table 6.1 shows that the p-values for the precision values are below the standard significant value, $\alpha = 0.05$. The reported figures show that, for most values of precision and recall, we can reject $\mathbf{H_{03}}$ and $\mathbf{H_{04}}$ in all but three cases (in bold in Table 6.1). Thus, we cannot claim to always reject $\mathbf{H_{03}}$ and $\mathbf{H_{04}}$: in two cases for VSM and one for JSM, recall values do not improve.

We use Cliff's d as introduced in Section 6.2.6 to measure the effect of Trustrace over single IR techniques. Table 6.1 shows that Trustrace has a large effect on the improvements in precision and recall values in 66%, medium in 19%, small in 9% of the improvements. Only in the case of *SIP*, *i.e.*, 6%, with only $Histrace_{bugs}$, Trustrace recall values decreased with one large and one small effect size. Overall, the obtained effect size values indicate a practical improvement with Trustrace.

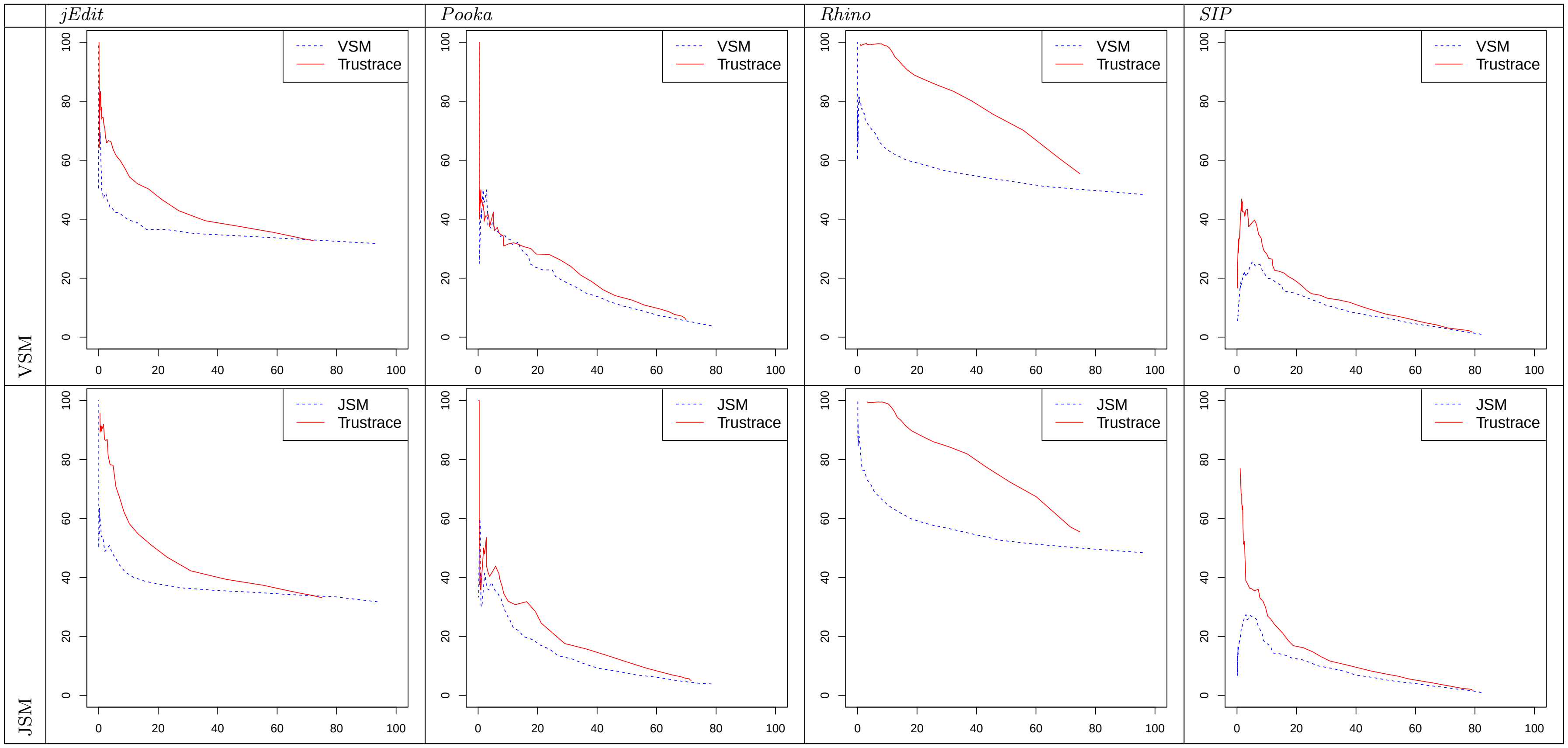


Figure 6.3 Precision and recall values of JSM, Trustrace, and VSM, with the threshold t varying from 0.01 to 1 by step of 0.01. The X axis shows recall and Y axis shows precision.

Table 6.1 Precision and recall values for *jEdit*, *Pooka*, *Rhino*, and *SIP*, Mann-whitney test results, and Cliff's d results

	Precision				**Recall**			
	VSM	$Trustrace_{VSM}$	p-value	Effect Size	**VSM**	$Trustrace_{VSM}$	p-value	Effect Size
jEdit	59.55	69.14	<0.01	0.59	5.81	8.31	<0.01	0.67
Pooka	42.28	52.46	<0.01	0.37	11.14	12.52	**0.99**	0.24
Rhino ($Histrace_{commits}$ *only)*	71.79	92.76	<0.01	0.92	6.82	9.52	<0.01	0.92
Rhino ($Histrace_{bugs}$ *only)*	71.79	93.73	<0.01	0.92	6.82	9.20	<0.01	0.92
Rhino	71.79	94.49	<0.01	0.84	6.82	12.33	<0.01	0.96
SIP, ($Histrace_{commits}$ *only)*	15.84	25.97	<0.01	0.61	15.61	15.79	<0.04	0.55
SIP, ($Histrace_{bugs}$ *only)*	15.84	42.97	<0.01	0.37	15.61	**11.07**	**<0.01**	**0.63**
SIP	15.84	24.28	<0.01	0.34	15.61	21.60	<0.01	0.48

	JSM	$Trustrace_{JSM}$	p-value	Effect Size	**JSM**	$Trustrace_{JSM}$	p-value	Effect Size
jEdit	52.82	71.12	<0.01	0.82	13.62	15.91	<0.01	0.88
Pooka	33.11	45.48	<0.01	0.47	13.33	16.45	<0.01	0.19
Rhino ($Histrace_{commits}$ *only)*	77.37	87.16	<0.01	0.15	15.56	16.18	<0.01	0.77
Rhino ($Histrace_{bugs}$ *only)*	77.37	90.88	<0.01	0.47	15.56	17.33	<0.01	0.59
Rhino	77.37	91.79	<0.01	0.47	15.56	18.26	<0.01	0.59
SIP ($Histrace_{commits}$ *only)*	15.83	21.29	<0.01	0.46	19.34	21.44	<0.01	0.60
SIP ($Histrace_{bugs}$ *only)*	15.83	37.94	<0.01	0.33	19.34	**14.24**	**<0.56**	**0.25**
SIP	15.83	27.67	<0.01	0.29	19.34	27.00	<0.01	0.92

We answer **RQ1:** How does the accuracy of the traceability links recovered by Trustrace compare with that of approaches based on JSM and VSM alone? as follow: Trustrace helps to recover more correct links than IR techniques alone. When two experts are available, Trustrace is always better. Only in one case, and with just a single expert, due to a lack of external source of information, recall went down.

RQ2: How does the accuracy of the traceability links recovered using DynWing compare to that using PCA?

When comparing DynWing with a PCA-based weighting technique, Figure 6.4 shows that DynWing provides better results than PCA. PCA tends to provide higher precision than recall in some cases whereas DynWing tends to find a better balance between precision and recall in all cases.

We performed a Mann-whitney test to analyse if DynWing statistically provides better F_1 scores or not. Table 6.2 shows that p-values are below the standard significant value $\alpha = 0.05$ for $Trustrace_{VSM}$. Thus, we reject $\mathbf{H_{05}}$. In the case of JSM, for *SIP*, using two experts does not yield any difference between DynWing and PCA-based weighting. Thus, we cannot reject $\mathbf{H_{06}}$ because in one case DynWing and PCA-based weighting provide the same results.

Thus, we answer **RQ2:** How does the accuracy of the traceability links recovered using DynWing compare to that using PCA? as follow: DynWing provides better weights for different experts than a PCA-based weighting technique. However, it is possible that in some cases PCA-based weighting provides the same (but not better) results as DynWing.

6.4 Discussion

We now provide qualitative analyses of our results and discuss observations from our empirical evaluation of Trustrace. We also return to our three conjectures.

6.4.1 Dataset Quality Analysis

Figure 6.3 shows that Trustrace has a better accuracy, on average, when compared to JSM and VSM for *jEdit* and *Rhino* but less for *Pooka* and *SIP*. We explain the differences in improvements by the many other factors that can impact the accuracy of traceability-recovery approaches, as discussed elsewhere (Ali *et al.* (2012a)). One of these factors is quality of source code identifiers. If there is a low similarity between the identifiers used by requirements and source code, then no matter how good an IR-based technique is, it would not yield results with high precision and recall values.

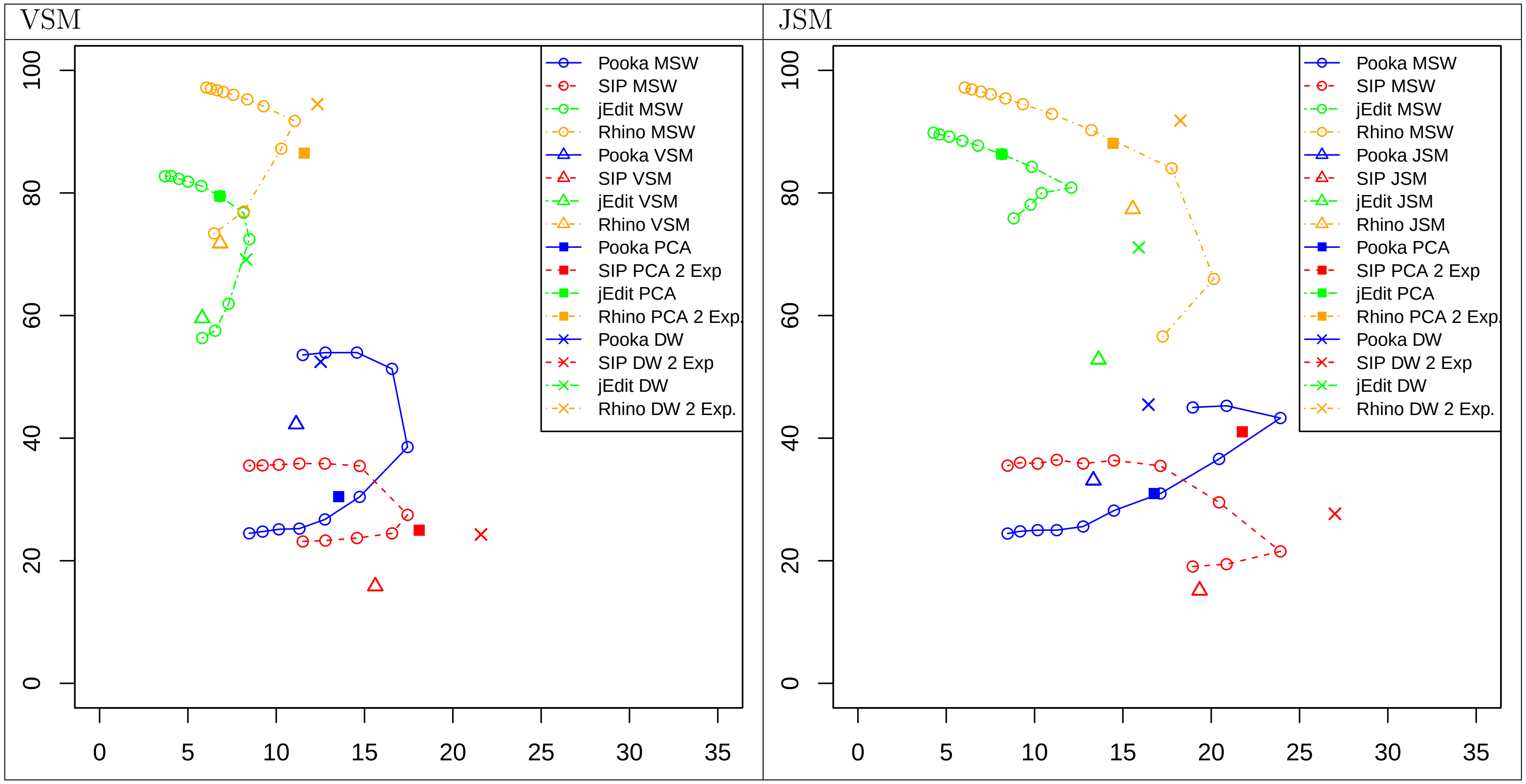


Figure 6.4 Precision and recall values, with the threshold t varying from 0.01 to 1 by steps of 0.01. The X axis shows precision values and Y axis recall values. DW represents the DynWing results.

To analyse the quality of the identifiers in our datasets and measure their similarity values, we use an approach proposed by Ali *et al.* (2011a), which helps to identify poor semantic areas in source code. We want to compute the similarity value between the set of requirements R, all merged into a single document $R_{all} = \bigcup_j r_j$, and the set of classes C, all merged into $C_{all} = \bigcup_j c_j$. We build the normalised term-by-document matrix to avoid any effect from the document lengths. Then, we use JSM and VSM to compute the similarity between $\bigcup_j r_j$ and $\bigcup_j c_j$. The similarity between these sets shows how close two documents are in terms of semantics.

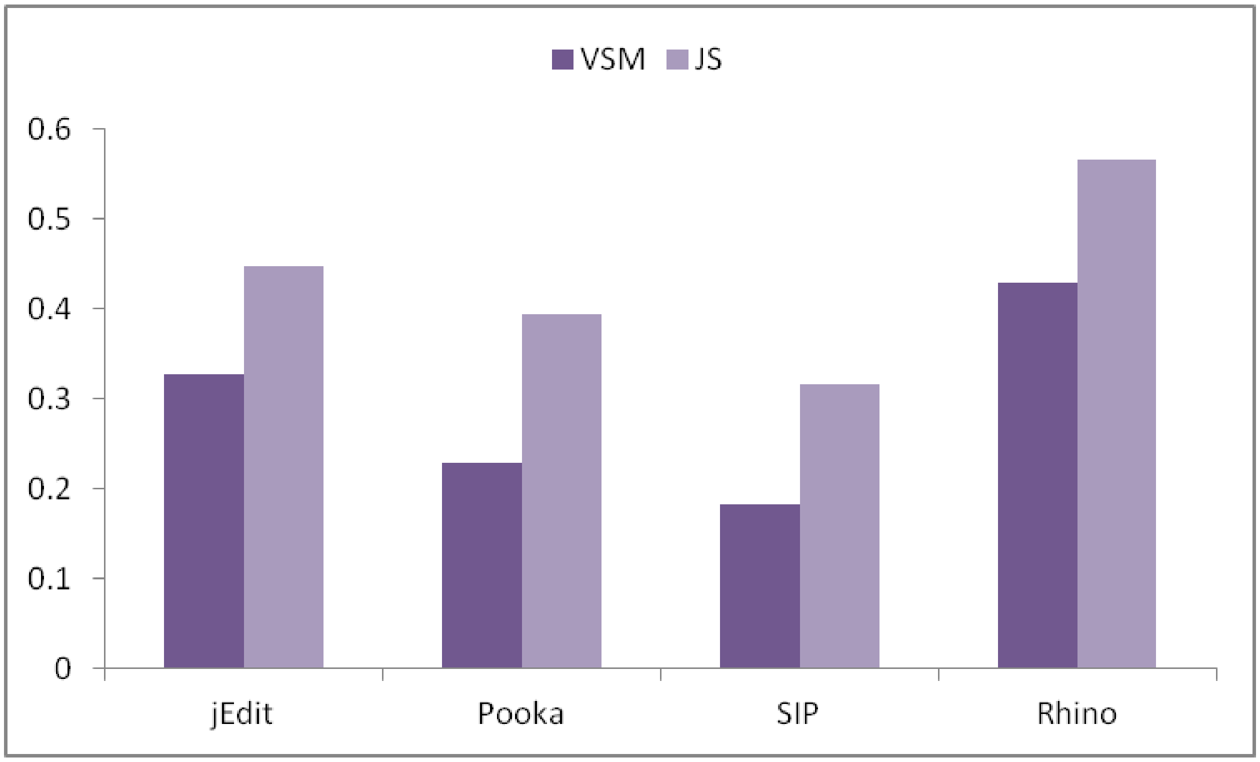


Figure 6.5 Similarity between merged requirements and source code documents

Figure 6.5 indeed shows that in the case of *Pooka* and *SIP* both JSM and VSM return low similarity values. Low similarity values among different documents would result into low precision and recall values for the traceability links (Ali *et al.* (2012a)). Figure 6.5 shows that JSM provides better similarity values between documents than VSM. Figure 6.3 shows the JSM tends to provide better precision and recall values than VSM.

A second factor from Ali *et al.* (2012a) is the knowledge of the developers who wrote the requirements. In our empirical evaluation, we recovered requirements for *Pooka* and *SIP* using Prereqir (Hayes *et al.* (2008)). All the requirements were written by subjects who use

Table 6.2 F_1 values for *jEdit*, *Pooka*, *Rhino*, and *SIP*, and Mann-whitney test results

	$Trustrace_{VSM}$								
	$Histrace_{commits}$			$Histrace_{bugs}$			$Histrace_{commits+bugs}$		
	DynWing F_1	PCA F_1	p-value	DynWing F_1	PCA F_1	p-value	DynWing F_1	PCA F_1	p-value
jEdit	8.56	7.50	<0.01	–	–	–	–	–	–
Pooka	10.52	7.42	<0.01	–	–	–	–	–	–
Rhino	12.18	11.61	<0.01	12.01	8.29	<0.01	36.91	10.55	<0.01
SIP	9.07	6.78	<0.01	6.57	6.51	<0.01	12.64	7.67	0.027
	$Trustrace_{JSM}$								
jEdit	9.60	8.49	<0.01	–	–	–	–	–	–
Pooka	8.02	6.60	<0.01	–	–	–	–	–	–
Rhino	13.21	12.33	<0.01	13.16	12.71	<0.01	40.72	15.18	<0.01
SIP	9.07	6.78	<0.01	8.11	6.15	0.02	11.39	7.84	0.127

e-mail clients and instant messengers on a regular basis, not by developers of such systems. Therefore, the subjects used non-technical terms in their requirements, which have little semantic similarity with the source code.

jEdit is an open-source Java text editor targeting the Java programming language. Although some *jEdit* users must be Java developers, we could not find any indication that all new feature requests were written by Java developers. Moreover, it is likely that jEdit users are not all Java developers. Users can ask for new features or report bugs in its online bug-tracking system. If a new requested feature is considered important by the community then it is included in a following version of *jEdit*. In this chapter, we considered such features requested by users as genuine requirements. Yet, users used technical terms to define requirements, which result into higher similarity values with the source code than *Pooka* and *SIP*.

In the case of *Rhino*, we used the ECMAScript specifications as requirements. ECMAScript specifications are detailed and written by technical writers. Thus, the similarity values between *Rhino* requirements and source code is higher than those of the other three datasets. We also observe that CVS/SVN commit messages are not always informative. Developers often did not provide relevant messages for the CVS/SVN commits or only some very generic message, *e.g.*, "changed", "updated files", and–or "fixed bug". Yet, the CVS commit messages of *Rhino* were better than those of the other three datasets.

Thus, we can say that high semantic similarity between requirements, source code, CVS/SVN commit messages, and bug reports affect the results of any RTA. However, Table 6.1 shows that even with low similarity values between documents, Trustrace improves the precision and recall values of the recovered links.

6.4.2 DynWing vs. MSW vs. PCA

Manually tuning the weights used to combine the opinions of experts allows favoring precision over recall or vice-versa. However, this manual tuning requires a labelled corpus, *i.e.*, an oracle, that is almost never available. Consequently, previous approaches (Ali *et al.* (2011b); Poshyvanyk *et al.* (2007)) used multiple-static weights (MSW) (see Section 5.4.1) to define a range of weights to tune their approaches. We analysed the links recovered when using MSW to compare DynWing with carefully manually-tuned weights.

We use MSW in Trumo, then compute the precision and recall of the obtained links in comparison to our four oracles: ($\text{Oracle}_{\text{jEdit}}$, Oracle_{Pooka}, $\text{Oracle}_{\text{R}hino}$, and Oracles_{SIP}) to find the optimal λ values for Equation (7.1) in the Trustrace model in Section 5.2, for *jEdit*, *Pooka*, *Rhino*, and *SIP*, respectively. We use different λ values to assess which λ value provides better results. We use $\lambda \in [0, 1]$ values with a 0.1 increment.

We observe in Figure 6.4 that DynWing is close to the optimal solution that MSW provides but, still, there is room for improvement in terms of precision and recall. For example, in the case of *Pooka*, with only $Histrace_{commits}$, DynWing (cross signs) provides on average 52.46 and 12.52 precision and recall values whereas MSW provides on average 51.31 and 14.63: DynWing increased the recall value by 2.11 at the price of a decrease in precision by 1.14. In the case of *Rhino* and *SIP*, we only show the average precision and recall of two experts in the graph for the clarity of the graph. Figure 6.4 and Table 6.1 show that using more than one expert provides better results than the MSW weighting technique. However, Figure 6.4 shows that for each system, the ranges of weights vary. Thus, we cannot identify one range of weights that would yield traceability links with reasonable precision and recall values for all the four systems. As the number of experts increases, it would become more difficult to identify the most appropriate weights or range of weights. Instead, DynWing would relieve project managers from choosing static weights.

We observe that treating each and every link independently helps to increase precision and recall. Some researchers (Poshyvanyk *et al.* (2007); Gethers *et al.* (2011)) performed several experiments using various weights to provide a range of weights that work well with their approaches/datasets. However, we cannot generalise such kinds of weights, because every dataset is unique (Ali *et al.* (2012a)). In addition, a range of weights is only a start to recover traceability links and, without an oracle, it is impossible to identify what weights and–or ranges of weights are suitable. Moreover, Figure 6.4 shows that we cannot provide a reasonable range using MSW in the cases of *Pooka* and *SIP*: the MSW weight range for JSM is not suitable for VSM. Thus, we cannot impose the same weight range on all the datasets and IR techniques as proposed by other researchers (Poshyvanyk *et al.* (2007); Gethers *et al.* (2011)).

Gethers *et al.* (2011) proposed a PCA-based weighting technique that does not require an oracle to define weights. Figure 6.4 shows that the PCA-based weighting technique favors precision over recall. Sometimes it provides better results than DynWing in term of precision only. DynWing increases both precision and recall.

This comparison of DynWing with MSW also supports the answer of **RQ2** that DynWing tends to provide a better trade-off between precision and recall. We conclude that DynWing does not require any previous knowledge of an oracle and that treating each link separately and assigning weights on a per-link basis provides better precision and recall values than other weighting techniques. The achieved improvements are statistically significant and, most of the times, with large effect sizes.

6.4.3 Number of Experts

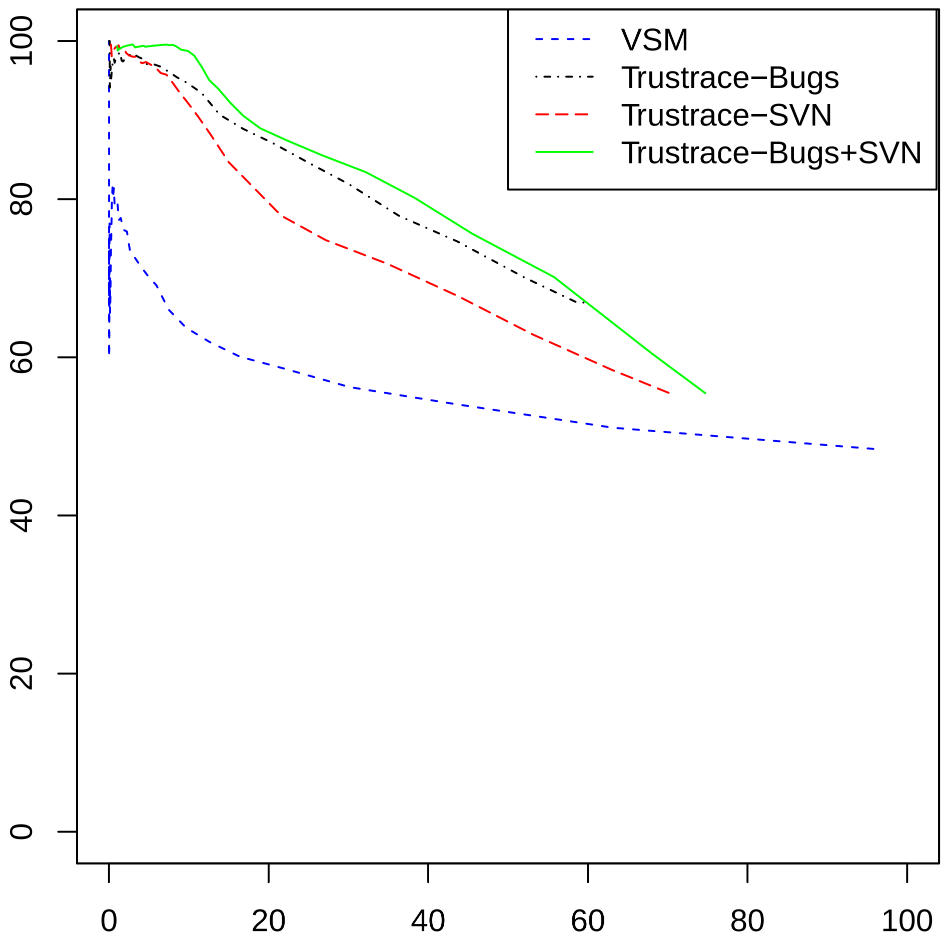


Figure 6.6 Rhino precision and recall graph for *Trustrace* using one and two experts. The X axis shows precision values and the Y axis recall values.

Figure 6.3 shows the results of two experts only, in the case of *Rhino* and *SIP*, for the sake of clarity. All the detailed figures and results are available online [1]. We only include one figure of two experts as an example in this chapter to show that adding more experts does impact precision and recall positively.

Figure 6.6 shows that using one expert provides better results than an IR-based technique alone. As we increase the number of experts, the graph shows a clear improvement in precision and recall. Table 6.1 reports that, in the case of *Rhino* and *SIP*, using two experts increases the precision and recall values more than using a single expert. It supports the idea that increasing the number of experts increases the precision and recall values.

6.4.4 Other Observations

The empirical evaluation supports our claim that Trustrace combined with IR techniques is effective in increasing the precision and recall values of some baseline requirements' traceability links. Our novel approach performs better than JSM and VSM.

While creating $\text{Oracle}_{\text{Pooka}}$ and $\text{Oracle}_{\text{SIP}}$, we tagged some requirements as unsupported features. While we performed the empirical evaluation, we found that Histrace produced links to some unsupported features. We manually verified these links and found that the source code related to these features indeed existed. We updated our oracles accordingly, which are the ones used in the previous sections.

For example, in *Pooka*, we declared the drag-and-drop requirement as an unsupported feature. Yet, Histrace produced some traceability links to this requirement. We manually verified these links and found that developers implemented the drag-and-drop feature partially but named it "dnd" while commenting "this is drag and drop feature" in some related SVN commit messages. This example shows that Trustrace can indeed help developers in recovering missing links from human mistakes and the limitations of automated approaches. In the cases of *jEdit* and *Rhino*, we did not find such missing links.

This observation shows that Histrace not only helps to recover traceability links but also may help in evolving traceability links: if a developer creates traceability links and, after some years, wants to update the traceability links, then she does not need to create/verify all the links again. She could run Histrace and–or Trustrace to obtain possible missing links that she can verify.

1. http://www.ptidej.net/download/experiments/tse12/

6.4.5 Practical Applicability of Trustrace

Trustrace enables practitioners to automatically recover traceability links between requirements and source code. The current model of Trustrace is general in the sense that all the conceptual steps of Trustrace, shown in Figure 6.1, can be changed. Indeed, Histrace could be customised to accommodate other software repositories, *e.g.*, mailing lists. Trumo could be applied to other problems, *e.g.*, feature location. For example, we recently customised Trumo to address bug location in combination to binary-class relationships. The most important aspect of Trustrace is that it does not require the tuning of some parameters for every dataset on which it is applied. DynWing is an automatic weighting scheme that assigns weights to the different experts at runtime. In contrast, a project manager would need to guess and assign weights to each expert.

The Trustrace model could be implemented in any software development environment. It does not require particular inputs or parameter tuning. Trustrace could mine any software repository and use them as experts to recover traceability links. A project manager could also use the output of Trustrace for other purposes than requirement traceability as well. For example, Trustrace could tell a project manager which requirements require more maintenance, in particular which requirements are causing more bugs and CVS/SVN commits.

6.4.6 Revisiting the Conjectures

In the introduction, we stated three conjectures regarding the use of other sources of information, of a trust model, and of a dynamic weighting technique to improve the accuracy of the requirement traceability links recovered using an IR technique.

Within the limits of the threats to the validity of the results of our empirical study, we conclude that our conjectures are true. Indeed, our empirical study shows that the requirement traceability links recovered through mining software repositories, *i.e.*, CVS/SVN repositories and Bugzilla bug-tracking systems, can be considered as experts whose opinions can be used in a trust model to discard/re-rank the links provided by an IR technique. The experts' opinions must be combined dynamically, *i.e.*, on a per-link basis, to reap the full benefits of the trust model.

To the best of our knowledge, this work is the first stating these conjectures and reporting on the benefits of combining software repositories to improve the accuracy of requirement traceability links. It is also the first use of a dynamic weighting technique in combination with a trust model. We expect that other software repositories could be useful during the traceability recovery process and that other models of trust and weighting techniques could even further improve the accuracy of the links recovered by an IR technique.

6.4.7 Threats to Validity

Several threats potentially impact the validity of our experimental results.

Construct validity:

We quantified the degree of inaccuracy of our automatic requirement traceability approach by means of a validation of the precision and recall values using manually-built oracles. To manually built the oracle or use pre-built oracles, we follow the same steps as described in Appendix A. We manually verified some of the automatically recovered links by our approach to discover any imperfection in manually-built oracles. We improved the oracles after applying Trustrace and discovering missing links.

Internal Validity:

The internal validity of our empirical study could be threatened by our choice of the λ value: other values could lead to different results. We mitigated this threat by studying the impact of λ on the precision and recall values of our approach in Section 6.3 and using MSW- and PCA-generated λ values. The global trust over an expert could also impact the validity of results. We mitigated this threat by using the same setting for all the experiments and found the same trends of improvements.

External Validity:

Our empirical study is limited to four systems, *i.e.*, *jEdit*, *Pooka*, *Rhino*, and *SIP*. Yet, our approach is applicable to any other systems. However, we cannot claim that the same results would be achieved with other systems. Different systems with different SVN commit logs, requirements, bug descriptions, and source code may lead to different results. Yet, the four selected systems have different SVN commit logs, SVN commit messages, requirements, bug reports, and source code quality. Our choice reduces the threat to the external validity of our empirical study.

Conclusion validity:

The appropriate non-parametric test, Mann-Whitney, was performed to statistically reject the null-hypotheses, which does not make any assumption on the data distribution. We also mitigated this threat by applying Shapiro-Wilk test to verify the distribution of our data points to select an appropriate statistical test and effect size.

6.5 Summary

In this chapter, we conjectured that: (1) we could mine software repositories to support the traceability recovery process; (2) we could consider heterogeneous sources of information to discard/re-rank the traceability links provided by an IR technique to improve its accuracy;

and, (3) we could use an automatic, per-link experts' weighting technique to avoid the need of manually-built oracles to tune weights.

To support our conjectures, we proposed a new approach, Trustrace, to improve the precision and recall values of some baseline traceability links. Trustrace is based on mining techniques, on a trust model, and a dynamic weighting technique. Trustrace consists of three parts, *i.e.*, Hisrace, Trumo (see Sections 4.1 and 5), and DynWing. Figure 6.1 shows that Trustrace indeed improves the accuracy of traceability links recovered by either JSM or VSM.

Trustrace is the first approach to integrate time information (from CVS/SVN commit logs) and bug information (from Bugzilla reports) to recover requirement traceability links. We applied Trustrace on *jEdit*, *Pooka*, *Rhino*, and *SIP* to compare its requirement-traceability links with those recovered using only the JSM or VSM techniques, in terms of precision and recall. We showed that Trustrace improves with statistical significance the precision and recall values of the traceability links. We also compared DynWing with a PCA-based weighting technique in term of F_1 score. We thus showed that our trust-based approach indeed improves precision and recall and also that CVS/SVN commit messages and bug reports are useful in the traceability recovery process.

We thus conclude that our conjectures are supported: the accuracy of the traceability links between requirements and source code recovered by an IR technique are improved by (1) mining software repositories and considering the links recovered through these repositories as experts, (2) using a trust model inspired by Web-trust models to combine these experts' opinions, and (3) weighting the experts' opinions on a per-link basis for each link recovered by the IR technique.

CHAPTER 7

ASSESSING THE USEFULNESS OF TRUMO FOR BUG LOCATION

Preliminary to any software maintenance task, a developer must understand the source code. Due to incomplete/missing documentation, software evolution, and software aging, developers may have difficulties to comprehend the source code. Program comprehension-related activities are involved in 50% to 90% of all software maintenance activities (Muller *et al.* (2000)). Thus, these difficulties to understand source code increase maintenance costs. A particular task during which these difficulties slow down developers is that of bug location. To fix a bug, developers must locate the bug in the source code of a program: they must identify the classes (and–or parts thereof) describing the unexpected behaviour of the program. Therefore, they must understand the source code enough to identify the culprit classes and modify them to fix the bug. Effectively automating this task could reduce maintenance costs by reducing developers' effort (Lukins *et al.* (2008)).

There exist several semi-automated bug-location techniques (BLTs) for OO programs (Antoniol *et al.* (2002b); Marcus and Maletic (2003a); Poshyvanyk *et al.* (2007); Salah *et al.* (2005)). These techniques typically analyse a bug report and the source code of a program to report a list of the classes ordered by their likelihood to be related to the bug. These techniques mainly divide into three sets: static, dynamic, and hybrid. To locate a bug, static BLTs (Antoniol *et al.* (2002b); Marcus and Maletic (2003a); Robillard (2008)) use textual information extracted using static analyses on the program source code whereas dynamic BLTs (Salah *et al.* (2005)) use textual information extracted from the execution traces of a program. Hybrid BLTs (Poshyvanyk *et al.* (2007)) use both static and dynamic analyses. In the following, we consider only static BLTs, *e.g.*, IR-based BLTs, because some bugs may prevent a program to compile or it may not be possible to retrieve execution traces from a program because it is not yet completed. Therefore, static BLTs have advantages over dynamic ones because they do not require a compilable program and can be applied at any stage of its development or maintenance.

To the best of our knowledge, no previous authors considered binary-class relationships (BCRs). BCRs include use, association, aggregation, composition, and inheritance. We conjecture that the existence of BCRs among classes is useful information to re-rank the list of culprit classes. Our conjecture stems from the observations that developers express the design and implementation of their programs in terms of classes and the BCRs among these. For example, the classes playing a role in implementing the "find/replace" feature in a program would probably be related through some BCRs. We therefore study the impact of using BCRs to improve the ranking of classes to help developers during their bug-location tasks. BCRs are extracted from source code using static analyses. We exclude from our study composition BCR that, in general, require dynamic analyses.

To test our conjecture, we propose LIBCROOS, an approach that uses LInguistic (textual) and BCRs of OO systems, to improve the accuracy of IR techniques. In LIBCROOS, an IR technique creates a set of so-called baseline links between a bug report and the classes of a program. Then, each BCR among the classes acts like an expert to vote on the baseline links. The higher the number of experts voting (Poshyvanyk *et al.* (2007); Ali *et al.* (2011b)) for a baseline link, the higher the confidence in the link, and the higher LIBCROOS puts that class in its ranked list. Thus, LIBCROOS is similar to trust-model (6.1), more sources of information, *i.e.*, BCRs, confirming a link the higher the trust would be on the link. LIBCROOS is a complementary approach to any IR technique, which is, to the best of our knowledge, the first approach to use BCRs as experts to vote on the links between bug reports and classes. We perform an empirical study to compare LIBCROOS with two IR techniques alone: LSI and VSM. We evaluate the effectiveness of our proposed approach on four programs—Jabref, Lucene, muCommander, and Rhino.

The remaining chapter is organised as follows. Section 7.1 describes proposed approach in details and sketches our implementation. Section 8.2 provides details on our empirical study. Section 8.3 reports the results, discussions, and threats to the validity of our findings. Finally, Section 8.5 provides the summary of the chapter.

7.1 LIBCROOS

We now present the details of our approach for bug location using BCRs, which uses textual information and BCRs extracted from the source code to link classes to bug reports. Figure 7.1 shows the high-level view of LIBCROOS. It has three main modules, *i.e.*, an IR engine, BCRTrace, and a $Trumo_{Ranker}$. We give some details about the abstract model of LIBCROOS and then explain each module.

7.1.1 LIBCROOS Abstract Model

We reuse some of the math symbols as defined in Section 4.2. Let us define four functions α, β, δ, and γ: the first function, $\alpha(b_i, c_j)$, returns the similarity score $\sigma_{i,j}$ between a class c_j and a bug report b_i computed by the IR engine. The function $\beta(b_i)$ returns the set L_i of classes linked to a bug b_i. The function $\delta(L_i, r_k)$ returns the set $Q_{i,k}$, and the function $\gamma(c_j, L_i, r_k)$ returns 1 if $c_j \in Q_{i,k}$ and 0 otherwise.

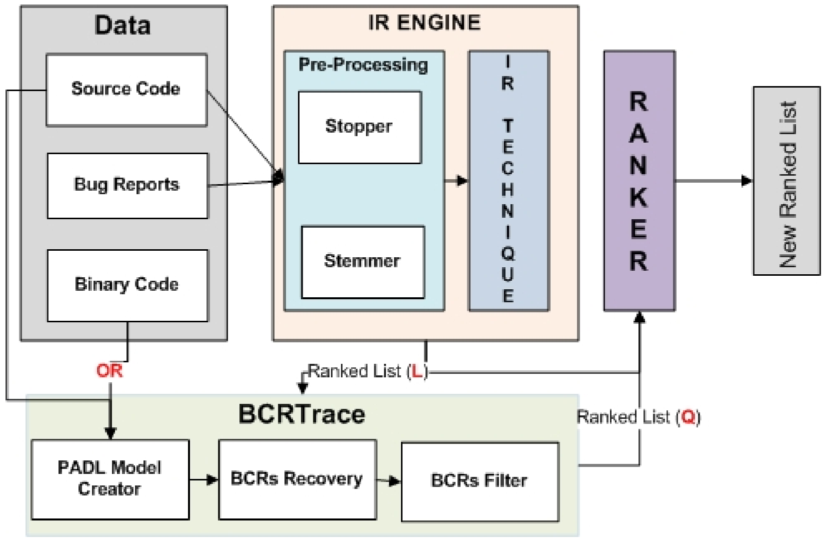


Figure 7.1 High-level Diagram of LIBCROOS

We also define $\psi_{i,j}$, a function that computes the final similarity between a class c_j and a bug b_i by combining the vote of each BCR, *i.e.*, inheritance, association, aggregation, and use relation, and IR technique, as:

$$\psi_{i,j} = \lambda \frac{\sum_{k=1}^{|R|} \gamma(c_j, L_i, r_k)}{|R|} + (1-\lambda)\,\sigma_{i,j} \tag{7.1}$$

where $\lambda \in [0,1]$, represents the weight of the IR technique. The higher the evidence, *i.e.*, number of BCRs ($\sum \gamma(c_j, L_i, r_k)$), the higher the new similarity $\psi_{i,j}$. In the contrary, little evidence decreases $\psi_{i,j}$ relatively to the similarities of other $c_j \in L_i$. LIBCROOS model is thus similar to the Trustrace model (see Section 6.1), the higher the evidence for a link, the higher the similarity value of the link.

7.1.2 IR Engine

LIBCROOS uses IR techniques as an engine to create links between bug reports B and classes C. LIBCROOS is not dependent on a particular IR technique; any IR technique could be used with LIBCROOS. IR techniques consider both bug reports and classes as textual documents. For source code, we use a parser as explained in Section 2.1.1, *e.g.*, a Java parser, to extract all source code identifiers. The parser discard extra information, *e.g.*, data types, from the source code (Ali *et al.* (2011a)). IR techniques only consider the textual information extracted from the bug reports and source code to compute the similarity between them. IR techniques compute the similarity, as explained in Section 7.1.2, based on the bug reports and source code terms and–or the distributions thereof. With any IR technique, a high similarity value between bug reports and source code suggests a potential link between them. The similarity between two documents is measured by the cosine of the angle between their corresponding vectors of terms' weight. The details about computing cosine is explained in Section 2.2.1. Thus, LIBCROOS IR engine generates a set $\mathcal{L}$.

7.1.3 Usage of BCRTrace

The BCRTrace is the part of LIBCROOS that provides the BCRs between classes. BCR-Trace uses Ptidej tool suit and its PADL meta-model to generate BCRs among classes. BCRTrace takes as input the set $\mathcal{L}$ provided by the IR engine and the source code or binary code of the program. It produces four experts, *i.e.*, $Q_{i,1}, Q_{i,2}, Q_{i,3}$, and $Q_{i,4}$ as inheritance, use, association, and aggregation respectively (see Section 4.2).

7.1.4 $Trumo_{Ranker}$

The ranker is an instance of Trumo model (see Section 5.2). The $Trumo_{Ranker}$ assigns weights to different BCRs and similarity computed by IR engine to re-rank the classes linked to a bug. The $Trumo_{Ranker}$ takes the set $\mathcal{L}$ generated by the IR engine and the set Q generated by the BCRTrace as input. It uses the function $\gamma(c_j, L_i, r_k)$ to get the total number of BCRs for each c_j. For example, if a Bug_1 is linked to a class c_1 and the BCRTrace indicates that $c_1 \in Q_{1,1}$, *i.e.*, there exist an inheritance relationship between c_1 and another class of L_1 while $c_1 \notin Q_{1,2}$, $c_1 \notin Q_{1,3}$, and $c_1 \in Q_{1,4}$, then the $Trumo_{Ranker}$ observes that the total number of relationships found for c_1 is 2, *i.e.*, $\sum\limits_{k=1}^{4} \gamma(c_j, L_i, r_k) = 2$. Then, the $Trumo_{Ranker}$ assigns weights to the similarities $\sigma_{i,j}$ computed by the IR engine and $\gamma(c_j, L_i, r_k)$ provided by the BCRTrace. For example, a developer could assign more weight, *i.e.*, $\lambda = 0.7$, $\gamma(c_j, L_i, r_k)$ to and less weight *i.e.*, $\lambda = 0.3$, to $\sigma_{i,j}$. Based on Equation 7.1, the $Trumo_{Ranker}$ computes the final similarity $\psi_{i,j}$ for each link and then re-ranks the classes linked to a bug b_i based on their similarities.

7.2 Empirical Study

We perform an empirical study on four programs and with two state-of-the-art IR techniques to assess the accuracy of our proposed approach for bug location. This study provides data to assess the accuracy, in terms of ranking, of the improvement brought by LIBCROOS over two "traditional" IR techniques, using LSI and VSM alone.

The *goal* of our empirical study is to evaluate the effectiveness of our novel approach for bug location against traditional LSI and VSM-based approaches. In addition, we analyse which BCR helps to improve IR technique accuracy the most. The *quality focus* is the ability of LIBCROOS to link a bug report to appropriate classes in the source code in terms of ranking (Poshyvanyk *et al.* (2007)). The *perspective* is that of practitioners and researchers interested in locating bugs in source code with greater accuracy than that of currently-available bug-location approaches based on IR techniques. The *objects* of our empirical study are four open-source programs, Jabref, Lucene, muCommander, and Rhino.

7.2.1 Research Questions, Hypothesis, and Variables

The research questions that our empirical addresses are:

***RQ1*:** Does LIBCROOS provide better accuracy, in terms of ranking, than IR techniques?

***RQ2*:** What are the important BCRs that help to improve IR techniques accuracy more than the others?

To answer our research questions, we perform four experiments on Jabref, Lucene, muCommander, and Rhino using LIBCROOS, LSI, and VSM. We use a measure of ranking (Poshyvanyk *et al.* (2007)) to measure the accuracy of proposed and IR techniques to answer our research questions. LSI and VSM return ranked lists of classes for each bug in descending order of the textual similarities between the bug report and the classes. LIBCROOS returns a similar ranked list in descending order of the similarities computed using Equation 7.1. If a culprit class is lower (has a higher rank) in the ranked list then a developer must spend more effort to reach this actual culprit class because she must assess more candidate classes to solve the bug. Thus, the higher is a culprit class in a ranked list (decreased rank), the less is the developers' effort and the more accurate is an approach.

For RQ1, we consider the four BCRs, *i.e.*, use, association, aggregation, and inheritance, to analyse how much LIBCROOS can decrease the rank of classes to put culprit classes closer to the top of the list and, hence, can decrease a developer's effort. Consequently, we apply LIBCROOS, LSI, and VSM approaches on the four programs seeking to reject the two null hypotheses:

H_{01}**:** There is no statistical difference in terms of ranking between LIBCROOS and VSM.

H_{02}**:** There is no statistical difference in terms of ranking between LIBCROOS and LSI.

For RQ2, we use LIBCROOS with only one BCR at a time to observe which relationship helps more than the other to put the culprit classes at the top. Consequently, we apply LIBCROOS, LSI, and VSM approaches on the four programs seeking to reject the two null hypotheses:

H_{03}**:** All the BCRs equally improve accuracy, in terms of ranking, over VSM.

H_{04}**:** All the BCRs equally improve accuracy, in terms of ranking, over LSI.

We use the approaches, *i.e.*, LIBCROOS, LSI, and VSM, as independent variables and the rankings of the approaches as dependent variables to empirically attempt rejecting the null hypotheses.

Objects:

We select the four open-source programs, Jabref, Lucene, muCommander, and Rhino because they satisfy several criteria. First, we select open-source programs, so that other researchers can replicate our experiments. Second, we avoid small programs that do not represent programs handled by most developers. Third, three of the programs have been used in previous studies by other researchers (Eaddy *et al.* (2008c); Dit *et al.* (2011a)) for possible comparisons. Finally, three programs, *i.e.*, Jabref, muCommander, and Rhino, come with independent, manually-built oracles, which mitigates some of the threats to the validity of our results. Only for Lucene, we recovered links between bugs and source code. The details about all the four datasets are available in Appendix A.

7.2.2 Preparing Datasets for Experiments

We now detail how we prepare the input data necessary to answer our research questions.

Oracles: We use previously built oracles to verify if a class linked to a bug is true positive or false positive link. Some researchers (Eaddy *et al.* (2008c); Dit *et al.* (2011a)) manually and–or semi-automatically created links between bug reports and methods to create oracles. In this experiment, we link bug reports to classes, thus we converted the oracles at class level. The details about the creation of oracles are explained in Appendix A.

Generating Corpora: We download the source code of Jabref v2.6, Lucene v3.1, muCommander v0.8.5, and Rhino v1.5R4.1 from their respective CVS/SVN repositories. We generate corpora of all the programs for IR techniques to link bug reports to classes. To generate corpora, we extract source code identifiers using a dedicated Java parser (Ali *et al.* (2011a)) to extract all source-code identifiers. The Java parser builds an abstract syntax tree (AST) of the source code that can be queried to extract required identifiers, *e.g.*, class, method names, and so on. Only for Lucene, we download bug reports from the JIRA bug repository. For all the other programs, we used the bug reports provided by the other researchers (Eaddy *et al.* (2008c); Dit *et al.* (2011a)). We only use long description of bug reports. We perform the pre-processing steps for bug reports and source code as described in 2.1.1 to remove the noise from the datasets.

7.2.3 Linking Bugs Reports and Classes using VSM

We use VSM (see Section 2.2.1) to index the corpora generated in the previous step. For each bug report, VSM generates a ranked list of classes. For example, if there are 10 bugs then there would be ten ranked lists. In VSM, bug reports and classes are viewed as vectors of terms. Different term weighting schemes can be used to construct these vectors. The most popular scheme is TF/IDF. Once all bug reports and source code documents have been represented in the vectors, we compute the similarities between bug reports and classes to link them.

7.2.4 Linking Bugs Reports and Classes using LSI

LSI (see Section 2.2.2) is an IR technique based on the VSM. For each bug report, LSI generates a ranked list of classes. In this study, we tried various values for k for LSI and $k = 200$ for Jabref, muCommander, and Lucene, and $k = 50$ for Rhino provide better accuracy than the other k values. Once all bug reports and source code documents have been represented in the LSI sub-space, we compute the similarities between bug reports and classes using cosine similarity as explained in Section 7.1.2.

7.2.5 Linking Bugs Reports and Classes using LIBCROOS

In LIBCROOS, we take the processed corpora as input to link the bug reports with classes. Any IR technique could be used in LIBCROOS to link bugs to classes, as base line to start the process. We use the LSI and VSM IR techniques. We use our BCRTrace to analyse the relationships between the classes of each bug's ranked list. We put the classes in four sets, *i.e.*, $Q_{i,1}$, $Q_{i,2}$, $Q_{i,3}$, and $Q_{i,4}$ (see Section 7.1.3), depending on the BCR by which they are linked to other classes in the ranked list. If a class does not have any BCR with other classes, we do not keep that class in any of the sets. Thus, each set is treated as a different ranked list. Indeed, we have four ranked lists for each bug. We combine all ranked list as described in 7.1.4 using Equation 7.1.

To select λ values, we simply assigned 0.1 weight to each BCR for the corpora, *i.e.*, Jabref, Lucene, muCommander, and Rhino, and both IR techniques. We observed that using $\lambda = 0.1$ for each BCR yield to a good accuracy of LIBCROOS on the corpora. Furthermore, we analysed that assigning between 0.07 to 0.15 λ value to each BCR statistically increases the accuracy. More experiments are required with other corpora to generalise the weights for each BCR. Choosing a weight is still an open research question (Poshyvanyk *et al.* (2007)). However, we observe that using weights between 0.07 to 0.15 for all the corpora decreases the ranking of culprit classes.

7.2.6 Analysis Methods

We performed the following analysis on the LIBCROOS, LSI, and VSM ranked lists to answer our research questions by attempting to reject our null hypotheses.

We do not use precision and recall, because, on the one hand, the IR techniques would link all the bug reports to all the classes and recall would always be equals to 100% and because, on the other hand, using a high threshold value (Poshyvanyk *et al.* (2007)) based on textual similarity and linking only some bug reports to classes would yield a high precision. Thus, to compare LIBCROOS, LSI, and VSM, we use the rank of the first related class to a bug report as a measure of accuracy (Poshyvanyk *et al.* (2007)). In addition, we cannot analyse with the help of precision and recall whether buggy classes are moved up in the ranked list using LIBCROOS.

To answer RQ1, we compare LIBCROOS generated ranked list with LSI and VSM generated ranked lists of each bug. To answer RQ2, we use LIBCROOS with only one BCR at a time to compare its ranked list with LSI and VSM's ranked lists. Each approach used the same bug reports and classes. All null hypotheses have been tested using a paired, nonparametric test Mann-Whitney test (2.5.1). We use a significance level of 95% for the entire statistical tests. We use the Mann-Whitney test to assess whether the differences in accuracy, in terms of ranking, are statistically significant between the LIBCROOS, LSI, and VSM. Mann-Whitney is a nonparametric test and, therefore, it does not make any assumption about the distribution of the data.

7.3 Results and Discussion

This section presents the results of our experiments.

Figure 7.2 shows the box plots of the accuracy measure, in terms of ranking, of LIBCROOS, LSI, and VSM applied on the corpora. We use manually built oracles to analyse the rank of actual culprit class in the ranked list generated by LIBCROOS, LSI, and VSM. For all the corpora, LIBCROOS assigns a lower rank to the culprit classes in terms of lower quartile, mean, median, standard deviation, and upper quartile. The results illustrated in Figure 7.2 provide a high-level view of the accuracy of LIBCROOS. The smaller boxes represent the decreases in rankings, *i.e.*, the culprit classes at the top in the ranked lists.

Table 7.1 reports results for LIBCROOS using all the BCRs and using only one BCR at a time. Results show that LIBCROOS statistically decreases the ranks of culprit classes up to 67%. For example, in the case of muCommander (VSM), on average across bug reports, LIBCROOS decreases the rank of culprit from 39.88 to 13.23.

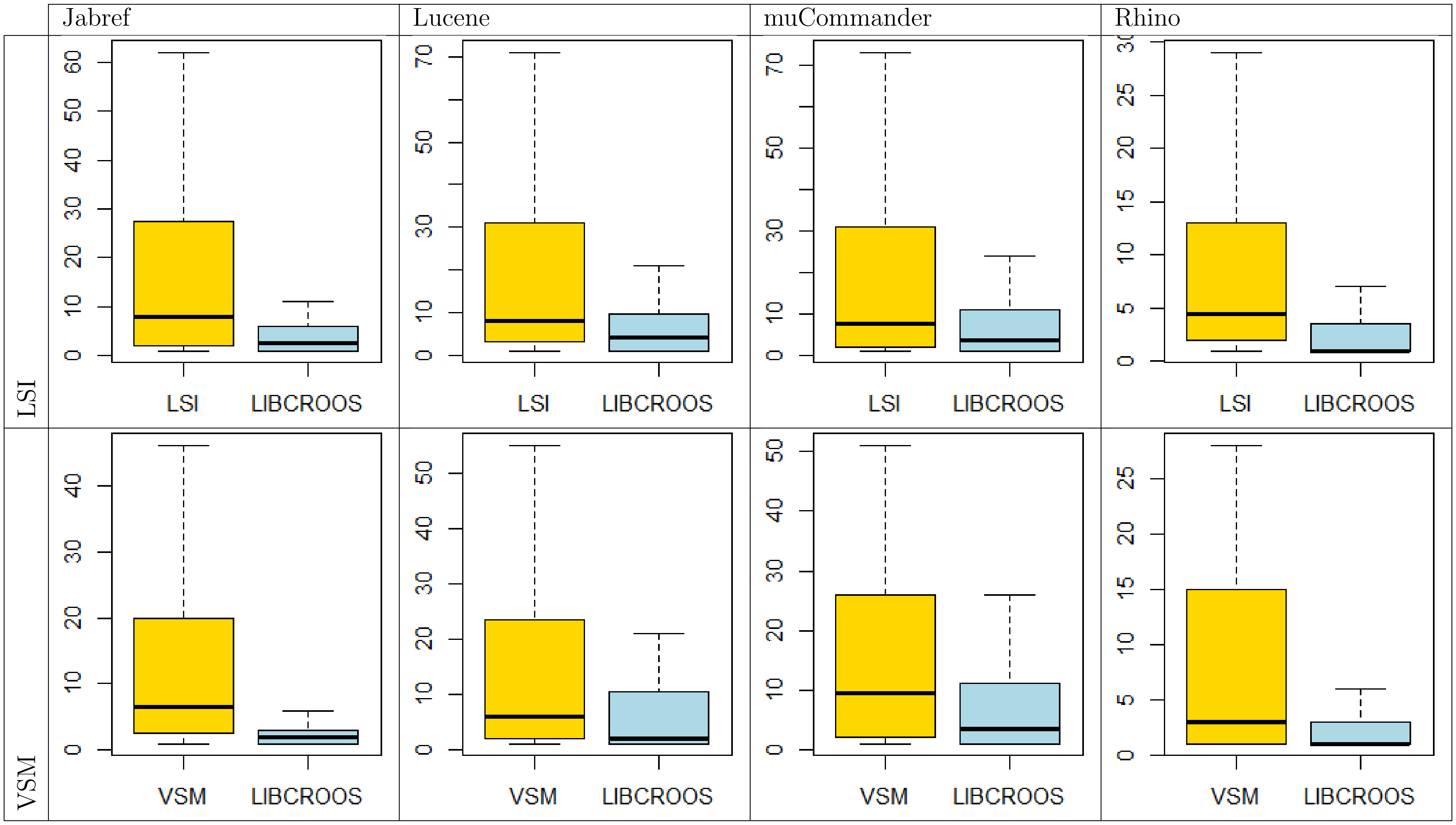


Figure 7.2 LIBCROOS, LSI, and VSM Results Boxplot Graph

Table 7.1 Descriptive statistics LIBCROOS, LSI, and VSM. LIBC. and SD represent LIBCROOS and Standard Deviation respectively

		LSI vs. LIBCROOS						VSM vs. LIBCROOS					
		Mean		Median		SD		Mean		Median		SD	
	Relationship	LSI	LIBC.	LSI	LIBC.	LSI	LIBC.	VSM	LIBC.	VSM	LIBC.	VSM	LIBC.
Jabref	ALL	22.19	5.47	8	2.5	32.86	7.59	16.14	2.5	6.5	2	23.30	2.09
	Aggregation	22.19	14.86	8	6	32.86	20.64	16.14	7.97	6.5	4	23.30	9.93
	Association	22.19	19.92	8	8	32.86	28.47	16.14	13.5	6.5	6.5	23.30	18.77
	Inheritance	22.19	10.72	8	4	32.86	15.33	16.14	5.75	6.5	3	23.30	7.59
	Use	22.19	19.56	8	8	32.86	27.997	16.14	12.97	6.5	6.5	23.30	18.35
Lucene	ALL	26.22	9.66	8	4	43.11	15.16	24.90	9.40	6	2	46.08	16.66
	Aggregation	26.22	16.42	8	6	43.11	25.47	24.90	15.47	6	5	46.08	28.74
	Association	26.22	24.90	8	7	43.11	40.50	24.90	23.88	6	6	46.08	43.57
	Inheritance	26.22	15.85	8	5	43.11	25.45	24.90	14.87	6	3.5	46.08	25.80
	Use	26.22	23.16	8	7	43.11	37.21	24.90	22.12	6	6	46.08	40.18
muCommander	ALL	36.72	11.59	7.5	3.5	89.33	24.96	39.88	13.23	9.5	3.5	105.20	32.01
	Aggregation	36.72	19.54	7.5	4.5	89.33	43.76	39.88	20.37	9.5	5.5	105.20	50.87
	Association	36.72	31.35	7.5	6.5	89.33	73.59	39.88	34.05	9.5	7.5	105.20	87.69
	Inheritance	36.72	20.08	7.5	5.5	89.33	46.26	39.88	22.21	9.5	5	105.20	55.64
	Use	36.72	29.94	7.5	6.5	89.33	69.59	39.88	32.69	9.5	8	105.20	83.66
Rhino	ALL	11.34	3.25	4.5	1	17.13	4.36	9.47	2.78	3	1	16.45	4.38
	Aggregation	11.34	6.97	4.5	3	17.13	9.55	9.47	5.47	3	2	16.45	8.76
	Association	11.34	9.41	4.5	3.5	17.13	14.66	9.47	7.72	3	2.5	16.45	13.87
	Inheritance	11.34	6.31	4.5	2.5	17.13	9.61	9.47	4.5	3	2	16.45	8.02
	Use	11.34	9.19	4.5	3.5	17.13	14.53	9.47	7.72	3	2.5	16.45	13.54

Standard deviation values show that all the ranks tend to be very close to the mean for LIBCROOS, whereas for LSI and VSM, they are spread out over a large range of values.

We have statistically significant evidence to reject the H_{01} and H_{02} hypotheses. The $p-$values, for all the comparison of LIBCROOS vs. LSI and LIBCROOS vs. VSM, are below the standard significant value, *i.e.*, 0.05. Results of RQ1 support our conjecture that adding BCRs with IR techniques helps to improve the accuracy, in terms of ranking, of IR techniques. Thus, we conclude that using BCRs among different classes that implement a same feature (or participate to a same bug) with IR techniques yields better accuracy than "traditional" IR techniques alone in bug location.

Thus, we answer RQ1 as follow: LIBCROOS helps to decrease the rank of culprit classes and put culprit classes higher in the ranked lists when compared to "traditional" IR techniques alone.

To answer RQ2, we use LIBCROOS with only one BCR at a time. We assign 0.1 weight to λ in Equation 7.1 (see Section 7.2.3. Table 7.1 shows the detailed results of each BCR separately. In the majority of the programs, we can see that inheritance is the most important relation, then aggregation, use, and association, for bug location. However, using other weights for each BCR could yield different results. We performed experiments using weights λ between 0.07 to 0.15 for each BCR and observed the same importance as mentioned before. We performed statistical tests to compare LIBCROOS using one BCR with LSI and VSM and test our null hypotheses H_{03} and H_{04}.

We have statistically-significant evidence to reject H_{03} and H_{04}. The $p-$values, for all the comparison of LIBCROOS using one BCR vs. LSI and vs. VSM, are below the standard significant value. The results show that using only one BCR also improves the accuracy, in terms of ranking, of IR techniques for bug location. LIBCROOS using only one BCR helps to decrease the ranks of culprit classes and put culprit classes higher in the ranked lists, when compared to "traditional" IR techniques alone. However, all the BCRs have different importance for bug location.

Thus, we answer RQ2 as follow: inheritance is the most important BCR to decrease the ranking, then aggregation, use, and association.

7.3.1 Threats to Validity

Several threats potentially limit the validity of our empirical study. We now discuss potential threats and how we control or mitigate them.

Construct validity: Construct validity concerns the relation between theory and observations. The degree of imprecision of automatic bug reports to class traceability link was quantified by means of a manual validation of the ranked lists generated by the approaches, using manually-built oracles. The oracles for three systems, *i.e.*, Jabref, muCommander, and Rhino, were built by other researchers. Thus, there is no bias in the links between bug reports and classes. All three corpora have been used by various researchers and manual oracles have been verified to avoid imprecision in the measurement. Only for Lucene, we automatically built the oracle by mining JIRA software repository. We used Lucene to mitigate the threat of imprecision of manually built oracles.

Internal Validity: The internal validity of a study is the extent to which a treatment effects change in the dependent variable. The internal validity of our empirical study could only be threatened by our choice of the λ value: other values could lead to different results. To mitigate this threat, we use the same λ for all the corpora and approaches. We used $\lambda = 0.1$ for every BCR in all the experiments. In addition, we used λ values between 0.07 to 0.15 to analyse; we found statistical improvement on these λ values too. However, it is possible using other λ values combination provide different results. Thus, more experiments are required.

External Validity: The external validity of a study relates to the extent to which we can generalise its results. Our empirical study is limited to four programs, Jabref, Lucene, muCommander, and Rhino. Yet, our approach is applicable to any other OO programs. However, we cannot claim that the same results would be achieved with other programs. Different programs with different usage patterns of BCRs, source code structure, and identifiers may lead to different results. However, the four selected programs have different source code size, different number of BCRs, and identifiers. Our choice reduces the threat to the external validity. The results suggest that inheritance helps better to put the culprit classes at the top. We explain this observation by the fact that inheritance is the most accurate BCR in the model because inheritance is explicit in the source code and actually expresses exactly what the developers want. For other BCRs that are not explicit in source code and recovered with the help of some algorithms, it may not always be the developers' intent to have these BCRs. Also, the analyses may miss some relationships or add more relationships than expressed in the design. Thus, more case studies are required to generalise our results.

Conclusion validity: Conclusion validity threats deals with the relation between the treatment and the outcome. The appropriate non-parametric test, Mann-Whitney, was performed to statistically reject the null-hypotheses, which does not make any assumption on the data distribution.

7.4 Summary

The literature (Antoniol *et al.* (2002b); Marcus and Maletic (2003a); Poshyvanyk *et al.* (2007)) showed that IR techniques are useful to link features, *e.g.*, bug reports, and source code, when a bug is defined as a feature that deviates from the program specification (Allen (2002)). However, IR techniques lack accuracy in terms of ranking (Ali *et al.* (2012a)). We conjectured that whenever developers implement some features, they use some BCRs among the different classes implementing the feature. Thus, combining BCRs with IR techniques could increase their accuracy in terms of ranking, to help developers with their bug location tasks. To verify our conjecture, we proposed a new approach, LIBCROOS that uses BCRs and textual information extracted from source code to link classes and bug reports. To the best of our knowledge, LIBCROOS is the first approach to use BCRs as experts to vote on the links recovered by an IR technique. We considered four commonly-used relationships, *i.e.*, use, association, aggregation, and inheritance, and two IR techniques, *i.e.*, LSI and VSM, to link classes and bug reports.

To evaluate the effectiveness of our proposed approach, we performed an empirical study on four software programs, *i.e.*, Jabref, Lucene, muCommander, and Rhino. We compared LIBCROOS-generated ranked lists of classes with the ranked lists produced by LSI and VSM alone. The results achieved in the reported empirical study showed that, in general, LIBCROOS improves the accuracy of LSI and VSM. In all experiments, LIBCROOS improved the accuracy of the IR-based technique and could reduce developers' efforts by putting actual culprit classes at the top in the ranked lists. We also used LIBCROOS with only one BCR at a time to analyse which BCR helps more to improve the accuracy of the IR techniques. In the majority of the programs, we observed that inheritance is a more important relation than aggregation, use, and association.

We also observed that, as the size of the source code increases, IR techniques produce larger ranked lists, which increase the difficulty of developers' bug-location tasks, because they must manually iterate through more potential culprit classes. LIBCROOS automatically increases the rank of non-culprit classes and brings actual culprit classes closer to the beginning of the ranked lists. Our results showed that the size of the source code does not impact the accuracy of LIBCROOS. Our empirical study helps developers and practitioners to understand how BCRs could be used for bug location. In addition, it suggests that developers must use more relationships among classes contributing to a same feature.

CHAPTER 8

IMPLEMENTATION OF COPARVO-BASED REQUIREMENTS TRACEABILITY

In this chapter, we present an approach, COde PARtitioning and VOting (COPARVO), to reduce the number of false positive links. COPARVO assumes that information extracted from different entities (*e.g.*, class names, comments, class variables or methods names) are different sources of information. COPARVO uses Partrace as an expert and voting model. In Partrace, each source of information may act as an expert recommending traceability links.

Our conjecture is that including all sources of information, *i.e.*, code partitions, may negatively impact precision and recall. Indeed when modifying a class, a developer much likely keeps the class API in line with the implemented feature while comments may not be updated. Thus, including outdated comments may cause to create many number of false positive links.

Example: Let us imagine that a developer is required to trace an email client requirement to its source code. The source code is in Java and requirements are written in English. We assume that the developer is using the IR technique, *i.e.*, VSM, to recover traceability links between source code and requirements. The developer is tracing a requirement Req_1 - "verify email address format before storing it in address book". Let us assume that the `EmailAddressFormatChecker` class is responsible to verify email address format and `AddAddressbookRecord` is responsible to store email addresses in the address book, whereas `SendEmail` sends out email. Let us further assume that in `SendEmail` an object `emailAddressFormatChecker` of type `EmailAddressFormatChecker` is created to verify email addresses before sending out the emails. In such situation, the developer risks to link Req_1 also to `SendEmail` because VSM will find matched terms, email, address, and format, between Req_1 and `SendEmail`. Thus, it is important to consider the position of the matched term in source code, *e.g.*, variable name versus comments.

Approach: OO source code is partitioned in four partitions, *e.g.*, class, method, variable names, and comments by Partrace 4.3. COPARVO merges all requirements in one file; each Source code partition is used to create a fictitious document containing all the text extracted from the source code related to the given partition. COPARVO considers four experts and thus four documents are created containing class names, methods names, variable names, and comments respectively. In the following, we will use as synonyms the terms partition or expert as both refers to the same concept.

The fictitious documents are used to compute similarity between requirements and the source code partitions. COPARVO ranks partitions based on the computed similarity; the top highest similar partitions are considered as trustable experts. If there are two or more partitions with the same textual similarity to requirements then we consider all of them as an expert. COPARVO then uses the identified experts to classify traceability links recovered with any IR technique. Each expert votes on the link to accept or reject it. To accept a link at least to two experts (or the majority of experts if there are three or more experts) must agree on a link, *i.e.*, have a non-zero computed similarity.

Validation: We apply COPARVO to reduce false positive links of a standard VSM with *TF/IDF* weighting scheme. We use three software systems: Pooka, SIP communicator, and iTrust. Our findings show that, in general, COPARVO improves accuracy of VSM and it also reduces up to 83% efforts required to manually remove false positive links.

The rest of the chapter is organised as follows: Section 8.1 describes proposed approach in detail and sketches our implementation of proposed approach. Section 8.2 presents the three case studies while Section 8.3 & 9.4 reports and discusses their results. Finally, Section 8.5 provides the summary of the chapter.

8.1 COPARVO

This section describes our proposed approach, *i.e.*, COPARVO, to improve the accuracy of IR-based approach and reduce effort of a project manager, in particular VSM, by partitioning source code. COPARVO is automated and supported by FacTrace (see Section 1.5), which provides several modules ranging from traceability recovery to traceability links verification.

We use Partrace (see Section 4.3) to partition the source code into four files. Each SCP acts as an expert to vote on the links recovered by an IR technique. We perform the pre-processing steps for requirements and source code as described in 2.1.1. Without the loss of generality, we use VSM (see Section 2.2.1) to index all the processed documents. In VSM each query is a requirement and documents are source code elements. Query and documents are viewed as a vector of terms. Once all the requirements and source code documents have been represented in the vectors, we compute the similarities between requirements and source code to link them.

8.1.1 Usage of Partrace

We revist the formal model as defined in Section 4.3. We are interested in finding the high similarity values between the set of requirements R, as merged into a single document

$R_{all} = \bigcup_j r_j$, and the document created by merging ψ_i projections $\bigcup_j \psi_i(C_j)$. The high cosine similarity shows the high trust over an expert. We order the high to low similarity experts to attain top two experts for voting. We use the cosine similarity between $\bigcup_j \psi_i(C_j)$ and requirement R_{all}. The highest similarities can be computed as:

$$\sigma_{max} = \max_i \{\sigma(R_{all}, \bigcup_j \psi_i(C_j))\}$$

where σ_{max} is the maximum attainable similarity and we are not interested in the absolute value rather in the rank associated to each projection ψ_i to identify most trustable experts and to select top two source-code experts β_i associated to the projections ψ_i for further processing. If two source code partitions have same similarity to requirements then we keep both of them. We consider two extreme cases: (1) *if the difference among source code partitions similarities is equal to or less than* 5% *then we consider both/all experts;* (2) *if difference between the first and second top source-code projection is equal to or greater than* 95% *then we only consider the first expert (top most information source).* The rationale of these two cases is that if the difference is equals to or less than 5%, then two or more partitions use similar semantics. The second rare case could occur if developers totally used different identifiers' names for two source code partitions; *e.g.*, coding a "send email function" and commenting it with "patient information" comments. Then, there would be high distance between method name and comments because they do not share any semantics. We consider at least the two top source-code partitions as experts for voting because if two partitions of a class are linking to a requirement then we can more likely trust the link.

There may be as many ψ_i functions as there are subsets of $\mathcal{C}$, however, for practical reasons we are interested in ψ_i that are (1) *arguably meaningful*; and (2) *easy to compute.* For example, a projection dividing classes into text documents corresponding to odd line numbers and even line numbers may be easy to compute but not meaningful. Thus, we consider the four projection: CN, VN, MN, CMT and combination thereof, *e.g.*, MN and CMT using both comments and API, which we denote $MN+CMT$ for the sake of simplicity. We use voting (see Section 5.3) to filter baseline links created by an IR technique.

8.2 Empirical Study

We perform an empirical study with three systems to assess the accuracy of our proposed approach for requirement traceability in term of F-measure. The empirical study provides data to assess the accuracy improvement over a traditional VSM-based approach and, consequently, the reduction of the project managers' effort brought to the maintainer when tracing requirements and validating traceability links to source code.

8.2.1 Goal

The *goal* of our case studies is to evaluate the effectiveness of COPARVO in improving accuracy of VSM and reducing effort required to manually discard false positive links. The *quality focus* is the ability of proposed approach to recover traceability links between requirements and source code in terms of F-measure 2.4.2. The *perspective* is that of practitioners and researchers, interested in recovering traceability links with greater F-measure value than currently-available traceability recovery approaches based on IR techniques. In addition, to remove as many as possible the false positive links.

8.2.2 Research Questions

The research questions that our empirical study addresses are:

RQ 1*: *How does COPARVO help to to find valuable partitions of source code that help in recovering traceability links?

RQ 2*: *How does COPARVO help to reduce the effort required to manually verify recovered traceability links?

RQ 3*: *How does the F-measure value of the traceability links recovered by COPARVO compare with a traditional VSM-based approach?

To answer these research questions, we assess the F-measure of proposed approach when identifying correct traceability links between requirements and source code on the one hand and between requirements and source code partitions on the other hand. Thus, we apply COPARVO and a VSM-based approach on the three systems seeking to reject the null hypotheses:

H_0: *There is no difference in the F-measure of the recovered traceability links when including whole source code or source code partitions selected by COPARVO.*

It is possible to formulate and accept an alternative hypothesis if the null hypothesis is rejected with relatively high confidence, which admits a positive effect of COPARVO on the retrieval accuracy:

H_a: *Recovering taceability links using COPARVO* significantly improves *the accuracy of the IR-based approach, in particular VSM.*

8.2.3 Variables and Objects

We use F-measure as independent variable and the COPARVO and VSM as dependent variables to empirically attempt rejecting the null hypotheses.

We select the three open-source systems, *Pooka*, *SIP*, and *iTrust*, because they satisfy several criteria. First, we select open-source systems, so that other researchers can replicate our experiment. Second, we avoid small systems that do not represent systems handled by most developers. Yet, all three systems were small enough so that we were able to recover and validate their requirements and traceability links manually in a previous work. Finally, their source code was freely available in their respective SVN repositories. Appendix A provides some descriptive statistics of the systems.

8.2.4 Usage of COPARVO

We now details how we gather and prepare the input data necessary to our empirical study.

Requirements: In a previous work (Ali *et al.* (2011c)), we used PREREQUIR (Hayes *et al.* (2008)) to recover requirements for Pooka and SIP. iTrust is complete dataset with source code, requirements, and traceability links matrix. We used these previously-built requirements to create manual traceability links between requirements and source code. To manually built the oracle or use pre-built oracles, we follow the same steps as described in Appendix A. The details about the requirements statistics are available in Appendix A.

Obtaining and Partitioning Source Code: We downloaded the source code of Pooka v2.0, SIP v1.0-draft, and iTrust v10.0 from their respective SVN repositories. Appendix A provides descriptive statistics of the three dataset of source code. We made sure that we could compile and run three systems by setting up the appropriate environments and downloading the relevant libraries before building traceability links. It helps us to analyse that we downloaded the correct version of the software with required libraries. However, it is not mandatory step.

We use Partrace 4.3 to partition source code in particular CN, MN, VN, and CMT and save them in respective files. The output of this step are the whole source code files, partitions of source code, *i.e.*, CN, MN, VN, CMT, and combination of partitions that we use for creating traceability links as explained in Sections 8.1. We perform pre-processing steps 2.1.1 on source code partitions and requirements for further processing.

Ascertaining Experts: First, we merge all the requirements for each system, in particular Pooka, SIP Comm., and iTrust, in one a file. Second, we merge all classes' CN, MN, VN, and CMT in four different files and name them as *cn.txt, mn.txt, vn.txt, and cmt.txt.* Lastly, we use VSM to compute similarity between merged requirement document and merged source code partitions file. Figure 8.1 shows the top experts (β_i) that has high similarities with requirements. For example, Pooka shows that MN and CMT must agree on each link. In the case of SIP Comm., we have extreme case (see Section 8.1, where difference between VN and CMT is less than 5%. Therefore, we will keep VN and CMT as second, MN as first expert.

Link Recovery and Voting Process: First, we use the VSM to create traceability links between requirements and whole source code ($\mathcal{L}$), which creates $11,056$, $79,422$, and $7,166$ links for Pooka, SIP, and iTrust, respectively.

Second, we link experts to requirements using the VSM. For example, in Pooka, MN and CMT are top experts. We use every class's MN to link them to requirements and same for CMT. Each set of traceability link between requirements and source code partition is stored in their respective category. These links will be used during expert voting schema. For example, we extract all the MN from `SpamSearchTerm.java` and `SpamFilter.java` and use VSM to link it to appropriate requirement, such as "*adding prelim support for spam filters*". We store this link in MN category. We repeat the same steps for CMT and store recovered links in CMT category.

Lastly, we use $\mathcal{L}$ as our base links and asserted experts vote on these base links. For example, in Pooka, if VSM recovers a link between R_1 and $Class_1$, then MN and CMT of $Class_1$ must link to R_1, if any of the compulsory expert does not link to R_1, we discard that link. We finally get the maximum similarity among $\mathcal{L}$ and the experts' links and update the VSM similarity value of that link.

8.2.5 Analysis Method

We performed the following analysis on the recovered links by our proposed approach, to answer our research questions and attempt rejecting our null hypotheses. We use $\text{Oracle}_{\text{Pooka}}$, $\text{Oracle}_{\text{SIP}}$, and $\text{Oracle}_{\text{iTrust}}$ to compute the F-measure values of the VSM and COPARVO. The VSM approach assigns a similarity value to each and every traceability link between requirements and whole source code, whereas COPARVO uses its own process (see Section 8.1) to create traceability links among requirements and ψ_{opt} source code sections.

We use a threshold t to prune the set of traceability links, keeping only links whose similarities values are greater than or equal to $t \in]0,1]$, as explained in 2.3. Then, we use the Mann-Whitney test (see Section 2.5.1) to assess whether the differences in F-measure values, in function of t, are statistically significant between the VSM and COPARVO. Mann-Whitney is a non-parametric test; therefore, it does not make any assumption about the distribution of the data.

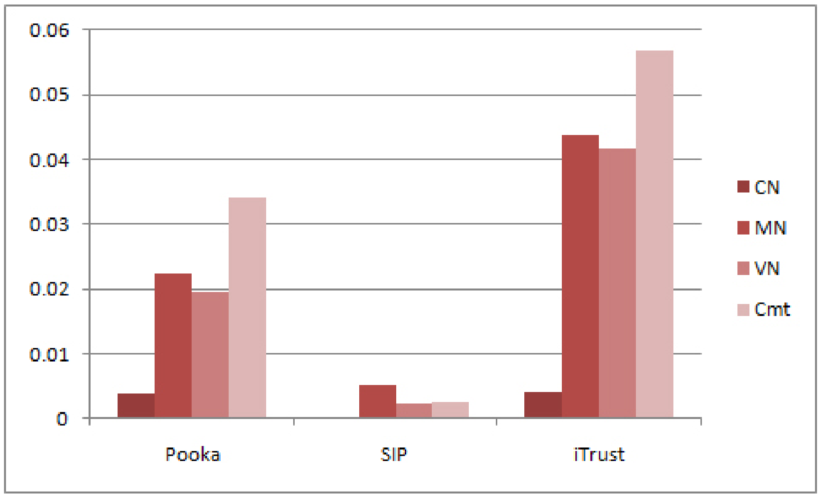


Figure 8.1 Top Experts for Traceability Link Recovery

8.3 Experiment Results

Figure 8.2 shows the F-measure values of VSM and COPARVO. It shows that β_i voting on recovered links provides better F-measures at different threshold t values.

Table 8.2 shows that COPARVO reduces project managers' effort from 39% to 83%. Table 8.1 shows that using different combination of source code partitions and whole source code provide poor results than proposed approach. SIP Comm. results shows that considering VN and CMT as second compulsory expert improves results.

We have statistically significant evidence to reject the H_0 hypothesis for all the three case studies. Table 8.2 shows that the p-values are below the standard significant value, $\alpha = 0.05$. We approve alternative hypothesis H_a. Table 8.1 shows that using other source code partitions than β_i also provide some time better precision, recall, and F-measure over VSM but not better than COPARVO. For example, in iTrust only using MN provides up to 19% F-measure but it decreases 8% precision than standard VSM, whereas COPARVO improves all precision, recall, and F-measure values.

We answer the RQ-1 as follow: partitioning source code and merging all of them and requirements in their respective files helps to find high similarity source code identifiers that can be considered as experts to vote on baseline links.

We answer the RQ-2 as follow: COPARVO uses expert voting scheme on baseline links to filter out false positive links. COPARVO helps to reduce effort, from 39% to 83%, required to manually remove false positive links.

We answer the RQ-3 as follow: source code partitions β_i statistically increases F-measure over the traditional way of recovering traceability links using whole source code.

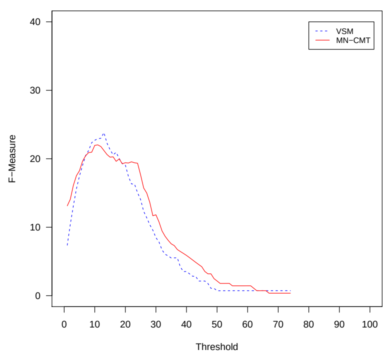
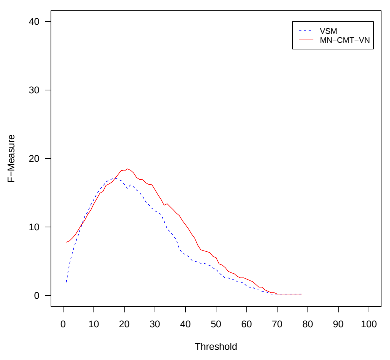
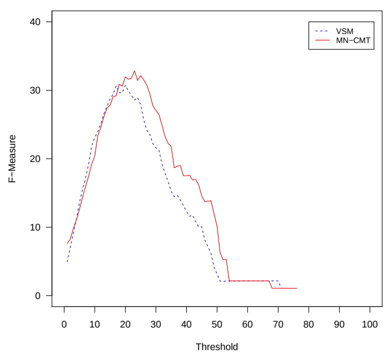
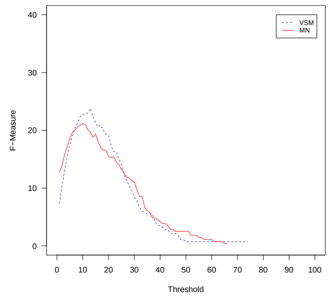
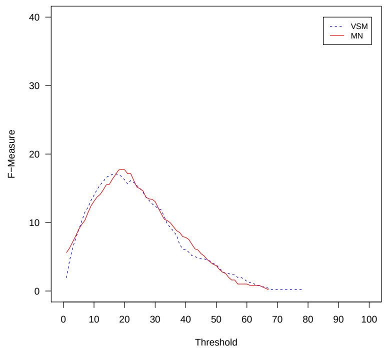
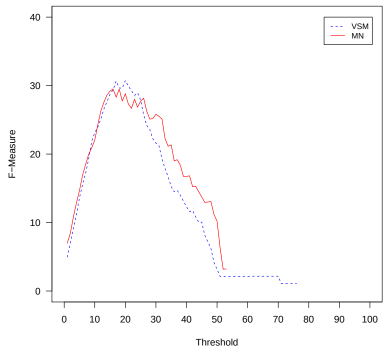
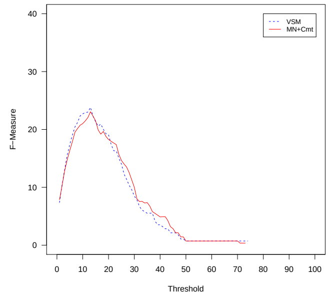
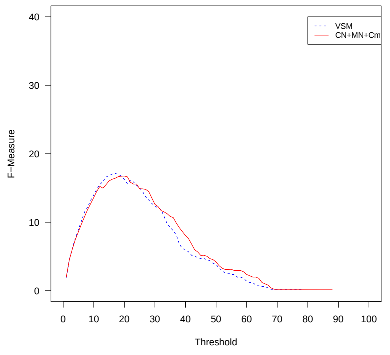
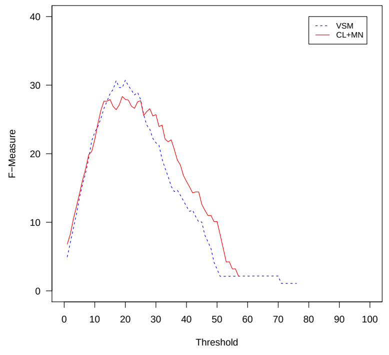
	Pooka	SIP	iTrust
COPARVO			
Single Partition			
Combinations			

Figure 8.2 F-measure values at different level of threshold. These graphs only shows the highest F-measure results in different categories for the sake of simplicity.

Table 8.1 COPARVO, VSM, and Other Source Code Partitions' Combination Results (Bold values represent the top experts (β_i) voting results). (+) sign represents the combination of source code partitions, whereas (-) sign represents different experts' voting on recovered links by VSM (COPARVO)

	Pooka			SIP Comm.			iTrust		
	Precision	Recall	F-Measure	Precision	Recall	F-Measure	Precision	Recall	F-Measure
VSM	42.28	11.14	8.19	14.10	13.25	7.17	49.03	20.38	12.87
MN-CMT	**43.22**	**11.69**	**9.34**	15.99	15.62	8.39	**57.14**	**23.05**	**14.77**
MN-VN-CMT	43.67	10.04	8.73	**17.12**	**14.64**	**8.70**	54.80	23.75	14.53
CN-MN-VN-CMT	40.99	7.16	8.12	16.61	11.33	8.15	42.10	22.04	15.10
CN	36.18	6.48	5.98	13.71	9.87	5.32	31.16	18.77	12.32
MN	42.22	10.75	8.28	17.54	15.87	8.57	41.57	28.63	19.87
VN	18.68	9.74	6.80	20.11	13.12	7.48	34.32	12.24	7.83
CMT	41.12	10.36	7.59	17.47	12.33	6.64	59.73	20.10	12.54
CN + MN	48.47	9.82	7.45	16.15	14.66	7.62	40.71	26.57	18.35
CN + VN	32.70	8.90	6.15	19.89	13.14	7.33	30.73	14.94	9.66
CN + CMT	39.16	10.92	8.04	19.05	12.18	6.49	54.84	20.68	12.79
CN + MN + VN	46.87	9.51	6.81	16.47	13.94	7.54	47.43	18.47	12.56
CN + MN + CMT	41.46	11.37	8.50	18.88	12.59	6.82	49.86	21.64	13.76
MN + VN	37.80	11.11	7.99	16.08	15.05	8.25	52.05	18.09	12.36
MN + CMT	40.67	11.46	8.52	19.06	12.57	6.84	52.56	21.44	13.66
VN + CMT	45.01	9.93	7.03	16.24	12.93	6.99	51.99	19.37	11.64

8.4 Discussion & Qualitative Analysis

We now discuss lesson learned from applying our approaches on the case studies.

8.4.1 Effort Reduction

COPARVO is completely automatic; it does not require additional effort from project manager. Table 8.2 shows the different number of traceability links recovered with VSM and COPARVO at $t = 0$. It shows that COPARVO helps to reduce by up to 83% the effort to manually discard false positive links. It also provides better precision, recall, and F-measure.

8.4.2 Ascertaining Experts

We recovered pre-requirements using PREREQUIR (Hayes *et al.* (2008)) for Pooka and SIP Communicator. Thus, the requirements of Pooka and SIP Communicator were not detailed and have less textual description, on average 15 words per requirement. iTrust, which comes with details requirements, has better textual similarity between requirements and source code partitions.

We only consider top two experts, more in the case of a tie, for voting. At least these two selected experts must agree on a link. The rationale behind this is that including lower quality expert for voting may impact results negatively. We performed more experiments to support our claim. Table 8.1 shows that as we include low quality expert in voting, it start decreasing the results.

8.4.3 Different Scenarios

To effectively evaluate our findings, we compare our proposed approach with three scenarios (i) *the fours experts (whole source code) are required to improve F-measure* (ii) *only one expert is sufficient to improve F-measure (single source code partition)*, and (iii) *simply removing low similarity source code sections can improve F-measure.* Below we discuss these scenarios in details:

The fours experts are required (Baseline): We included all identifiers, in particular CL, MN, VN, and CMT, to recover traceability links using VSM. Table 8.1 shows the results in the first section of table. We also used these results as baseline to compare other two scenarios to measure the accuracy improvement.

Single Source Code Partition: We split each Java file in four partitions: CL, MN, VN, and CMT. We used each partition and combination of the partition to recover traceability links. Table 8.1 shows that in SIP Communicator and iTrust results, only including MN increases

Table 8.2 COPARVO and VSM Total Recovered Links at 0 threshold and p-values of F-Measure

	VSM	**COPARVO**	**Effort**	**p-Value**
	Recovered Links	Recovered Links	**Saved**	
Pooka	11,056	4,514	59%	<0.01
SIP Comm.	79,422	13,271	83%	<0.01
iTrust	7,166	4,384	39%	<0.01

precision, recall, and F-measure over baseline results. However, comparing to COPARVO, none of the result using single source code partition has all three values, in particular precision, recall, and F-measure, better than COPARVO.

Combinations of Source Code Partitions: We performed this step to evaluate how much ascertaining experts and their voting help to improve F-measure in comparison to removing low similarity source code partitions to improve F-measure. For example, we only included MN and CMT source code partitions of Pooka to recover traceability links. The fourth part of Table 8.1 shows that it improves F-measure and recall, but decreases precision, when compared to baseline. Whereas, source code partitions combination provides lower precision, recall, and F-measure when compared to COPARVO. This comparison shows that only removing low similarity source code partitions does not help to improve precision, recall, and F-measure. It is also important that each link recovered by VSM must be voted by at least two top experts recovered by COPARVO.

8.4.4 Role of Identifiers

In classical IR-based approaches all the terms extracted from an artifact are used to define the semantics of the artifact (Capobianco *et al.* (2009)). In COPARVO we partition source code in four parts and measure the role of each partition on RT. The third section of Table 8.1 shows that developers usually integrate requirements concepts in MN. The second important part of source code is comments that play an important role to recover traceability links. However, in the case of SIP Communicator, CMT does not have much superiority over VN.

Thus, COPARVO also helps experts to find out which parts of the solution domain has high distance with problem domain. It also helps to improve the quality of identifiers as well as software quality that can help in program comprehension tasks. The similarity level among source code partitions and requirements shows which partition has more distance or low similarity to requirements. For example, in the case of SIP communicator, the similarity is very low between source code parts and requirements that results in poor quality links and huge number of recovered traceability links (see Table 8.2). It shows that developers are not using requirements terminology while implementing them in source code. Document size of requirements could also be the reason behind this low similarity.

Figure 8.1 shows that developers almost never used requirements' term to define a CN in SIP Communicator. Therefore, similarity between requirements and CN is near 0. They implemented requirements concepts at method level. iTrust has high similarity between source code partitions and requirements and it is well documented. Therefore, it provides better precision, recall, and F-measure using VSM and higher improvement after applying COPARVO.

8.4.5 Threats to Validity

Several threats potentially limit the validity of our experiments. We now discuss potential threats and how we control or mitigate them.

Construct validity: In our empirical study, we used widely adopted metrics, precision, recall, and F-measure, to assess the IR technique as well as their improvement. The oracle (traceability matrix) used to evaluate the tracing accuracy could also impact our results. To mitigate such a threat, two Ph.D. students created manual traceability oracles for Pooka and SIP Communicator and then a third Ph.D. person verified to avoid imprecision in the measurements. Moreover, we used iTrust traceability oracle developed by the developers who did not know the goal of our empirical study.

External Validity:

Our case studies are limited to three systems, Pooka, SIP, and iTrust. It is not comparable to industrial projects, but the datasets used by other authors (Antoniol *et al.* (2001); Lucia *et al.* (2007); Abadi *et al.* (2008)) to compare different IR methods have a comparable size. However, we cannot claim that the same results would be achieved with other systems. Different systems with different identifiers' quality ,reverse engineering code approach, requirements, using different software artifacts and other internal or external factors (Ali *et al.* (2012a)) may lead to different results. However, the three selected systems have different source code quality and requirements. Our choice reduces this threat to validity.

Conclusion validity:

We paid attention not to violate assumptions made by statistical tests. Whenever conditions necessary to use parametric statistics did not hold, *e.g.*, assumption on the data distribution, we used nonparametric tests, in particular Wilcoxon test for paired analysis. Wilcoxon does not make any assumption on the data distribution.

8.5 Summary

We proposed the use of source code partitioning to improve the performances of IR-based traceability recovery approach, in particular of VSM. Source code partition and merging all partitions and requirements were used by COPARVO to ascertain experts. We kept only top two experts, more in the case of tie, for voting on the recovered links by VSM. Our conjecture is that including lower similarity source code partitions can impact precision and recall.

The results achieved in the reported case study demonstrated that, in general, the proposed approach improves the retrieval accuracy of VSM. COPARVO could produce different results on different datasets. However, we used three systems that mitigate this threat. In all three systems, COPARVO improved the accuracy of IR-based approach and reduces efforts required to manually remove false positive links.

We analysed that as the source code size increases, IR-based approach recover more links. It makes a project managers' job difficult, because she must manually remove false positive links. The proposed approach automatically removes false positive links and reduces the effort between 39% and 83%. Results show that manual effort reduces as the number of recovered links increases.

We traced pre-requirements that were recovered using PREREQIR (Hayes *et al.* (2008)) for Pooka and SIP Communicator. The quality of pre-requirements is low as it provides the basic definition of users' needs. In the case of iTrust, requirements were properly documented and they were post-requirements. Therefore, we can see a high improvement in precision, recall, and F-measure using COPARVO. Using quality identifiers' names (see Section 9.4) and requirements will provide better results using COPARVO.

CHAPTER 9

USING DEVELOPERS' KNOWLEDGE FOR IMPROVING TERM WEIGHTING SCHEME

The researchers have shown and discussed (Ali *et al.* (2012a)) that IR-based techniques have different levels of accuracy depending on their parameters and on the systems on which they are applied. Thus, they agreed that there is still room for improving the accuracy of IR-based techniques, in particular by understanding how developers create/verify RT links (Ali *et al.* (2012a); Wang *et al.* (2011)). We could use developer's knowledge (see Section 4.4) as extra source of information to improve the accuracy of existing IR techniques. Thus, in this chapter, *our conjecture is that understanding how developers verify RT links can allow developing an improved IR-based technique to recover RT links with better accuracy than previous techniques.*

We bring evidence supporting our conjecture by stating and answering the following research questions:

***RQ1*:** What are the important source code entities[1] (SCEs) on which developers pay more attention when verifying RT links?

By answering the RQ1, we obtain a ranked list of SCEs, ranked based on the average amount of developers' visual attention on each SCE. That list shows that developers pay more visual attention on method names then comments then variable names and, finally, class names. Using this list, we build a new weighting scheme, SE/IDF, which integrates developers' preferred SCEs during RT links verification task. We compare this new weighting scheme against the state-of-the-art scheme, TF/IDF, and ask the following RQ2:

***RQ2*:** Does using SE/IDF allow developing a RT links recovery technique with a better accuracy than a technique using TF/IDF?

We observe in the ranked list of developers' SCEs that developers tend to prefer domain concepts over implementation concepts when verifying RT links. Indeed, while verifying RT links, developers pay more visual attentions to SCEs (class names, method names, and so on) representing domain concepts than those representing implementation concepts. Domain concepts are concepts pertaining to the system use. For example, in the Pooka e-mail client, `addAddress`, `saveAddressBook`, and `removeAddress` in `AddressBook.java` are

1. In this chapter, we call "source code entities" the class names, method names, variable names, and comments.

domain-related identifiers. Implementation concepts relate to data structures, GUI elements, databases, algorithms, and so on. Usually, developers use implementation concepts to recall the names of data types or algorithms. For example, in the Pooka e-mail client, `addFocusListener`, `updateThread`, and `buttonImageMap` in `AddressEntryTextArea.java` are implementation concepts. Using this difference between domain and implementation concepts, we build another weighting scheme, DOI/IDF, which distinguishes between domain and implementation related SCEs. We compare this new weighting scheme against the state-of-the-art scheme, TF/IDF, and ask the following RQ3:

***RQ3*:** Does using DOI/IDF allow developing a RT links recovery technique with a better accuracy than a technique using TF/IDF?

To answer these questions, we conduct an empirical study on RT consisting of two controlled experiments. The goal of the first experiment, to answer RQ1, is to observe developers' eye movements using an eye-tracker while they verify RT links to identify important SCEs. We perform this experiment with 26 subjects. We collect and analyse the data related to the developers' visual attention on SCEs. If developers pay more visual attention on a SCE, we consider it an important SCE.

The goal of the second experiment, to answer RQ2 and RQ3, is to measure the accuracy improvement of a LSI-based RT links recovery technique using SE/IDF and DOI/IDF over one using TF/IDF. We use LSI because it has been shown to produce interesting results (Marcus and Maletic (2003c)). We also use LDA, as used in previous work (Abebe and Tonella (2011); Maskeri *et al.* (2008)), to distinguish SCEs related to domain or implementation concepts.

We create three sets of RT links using LSI: $LSI_{TF/IDF}$, $LSI_{SE/IDF}$, and $LSI_{DOI/IDF}$, to compare them. Results show that, on iTrust and Pooka, $LSI_{DOI/IDF}$ statistically improves precision, recall, and F-measure on average up to 11.01%, 5.35%, and 5.03%, respectively.

The rest of the chapter is organised as follows: Section 9.1 and 9.2 provide the details of our experiment with the eye-tracker to understand developers' preferred SCEs the results. Sections 9.1 and 9.3 describe our novel weighting schemes, SE/IDF and DOI/IDF, sketches our implementation, and the results. Section 9.4 reports the discussion on our findings. Finally, Section 9.6 provides the summary of the chapter.

9.1 Empirical Study

Goal: Our empirical study consists of two controlled experiments. The goal of our first experiment is to identify developers' preferred SCEs using an eye-tracker during RT links verification process. In particular, we want to observe precisely which SCE receives more visual attention while verifying RT links. In our second experiment, our goal is to improve an IR-based RT links recovery technique by proposing novel weighting schemes based on developers' preferred SCEs, as observed in the eye-tracking experiment.

Study: In our first experiment, we use an eye-tracker to capture developers' visual attention. In our second experiment, we use the observations from the first experiment to design two novel weighting schemes, *i.e.*, SE/IDF and DOI/IDF. We perform experiments using both schemes combined with LSI to measure the improvement of these schemes when compared to TF/IDF weighting scheme.

Relevance: Understanding the importance of various SCEs is important from the point of view of both researchers and practitioners. For researchers, our results bring further evidence to support our conjecture that all SCEs must be weighted according to their importance. For practitioners, our results provide concrete evidence that they should pay attention to identifier names to help developers in performing RT links recovery. In addition, using more domain terms in source code would improve the accuracy of any IR-based technique to create RT links.

Hypotheses: We formulate the following null hypotheses:

$\boldsymbol{H}_{01}$**:** All SCEs have equal importance for developers.

$\boldsymbol{H}_{02}$**:** Domain and implementation related SCEs have equal importance for developers.

$\boldsymbol{H}_{03}$**:** There is no difference between the accuracy of a LSI-based technique using TF/IDF and using the novel weighting scheme, SE/IDF in terms of F-measure.

$\boldsymbol{H}_{04}$**:** There is no difference between the accuracy of a LSI-based technique using TF/IDF and using the novel weighting scheme, DOI/IDF in terms of F-measure.

$\boldsymbol{H}_{01}$ and $\boldsymbol{H}_{02}$ are related to RQ1 while $\boldsymbol{H}_{03}$ is related to RQ2 and $\boldsymbol{H}_{04}$ is related to RQ3. In Section 9.2, we address $\boldsymbol{H}_{01}$ and $\boldsymbol{H}_{02}$ while in Section 9.3, we address $\boldsymbol{H}_{03}$ and $\boldsymbol{H}_{04}$.

9.2 Experiment Design: Eye-Tracking

Eye-trackers are designed to work with the human visual system, it provides the focus of attention during the cognitive process of a human (Duchowski (2002)). An eye-tracker provides us with two main types of eyes-related data: fixations and saccades. We use fixations measure the subjects' visual attention because comprehension occurs during fixations (Rayner (1998); Duchowski (2007)). This choice directs our experiment settings.

9.2.1 Eye-tracking System

We use FaceLAB from Seeing Machine (see Section 4.4) which is a video based remote eye-tracker. Figure 9.1 shows the excerpt of the FaceLAB eye-tracker that we used in this study. We use two 27" LCD monitor for our experiment: the first one is used by the experimenter to set up and run the experiments while monitoring the quality of the eye-tracking data. We use the second one (screen resolution is 1920 x 1080) for displaying the Java source code and the questions to the subjects.

9.2.2 Experiment Settings

We show fragments of source code to developers and a requirement to verify RT links. We make sure that all the SCEs are easy to read and understand to avoid any bias. We define an Area of Interest (AOI) as an rectangle that encloses one (and only one) type of SCE. We establish two sets of AOIs for our source code stimulus. The first set contains four SCEs (class name, method name, variables, and comments) and we use this set to analyse which SCE is more important than the others for subjects. For the second set, each SCE could belong to either domain or implementation concept. For each AOI, we calculate time by adding the duration of all fixations available in that AOI. Eye-tracker provides us the duration of each fixation in milliseconds. The total time of fixation shows the amount of time spent by each subject on specific AOI while s/he focused on that part to understand it. The eye-tracker captures the fixations at the granularity of line. Therefore, for each identifier, we consider the line that contains the identifier. We divide the total calculated time spent on a SCE by that specific source code lines of code (LOC) to have the average time for each SCE. For example, if the comments are written on two lines and a subject spends 40 milliseconds to read comments, we divide 40 by 2 to get the time spent on each comment line.

9.2.3 Subjects Selection

There are 26 subjects from École Polytechnique de Montréal and University de Montréal. Most of the subjects performed traceability creation/verification tasks in the past; they are representative of junior developers, just hired in a company. The subjects are volunteers and they have guaranteed anonymity and all data has been gathered anonymously. We received the agreement from the Ethical Review Board of École Polytechnique de Montréal to perform and publish this study. There are 7 female and 19 male subjects. Out of 26 subjects, there are 4 masters and 22 Ph.D. students. All of the subjects have, on average, 3.39 years of Java programming experience. The subjects could leave experiment at any time, for any reason, without any kind of penalty. No subject left the study and it took on average 20 minutes to

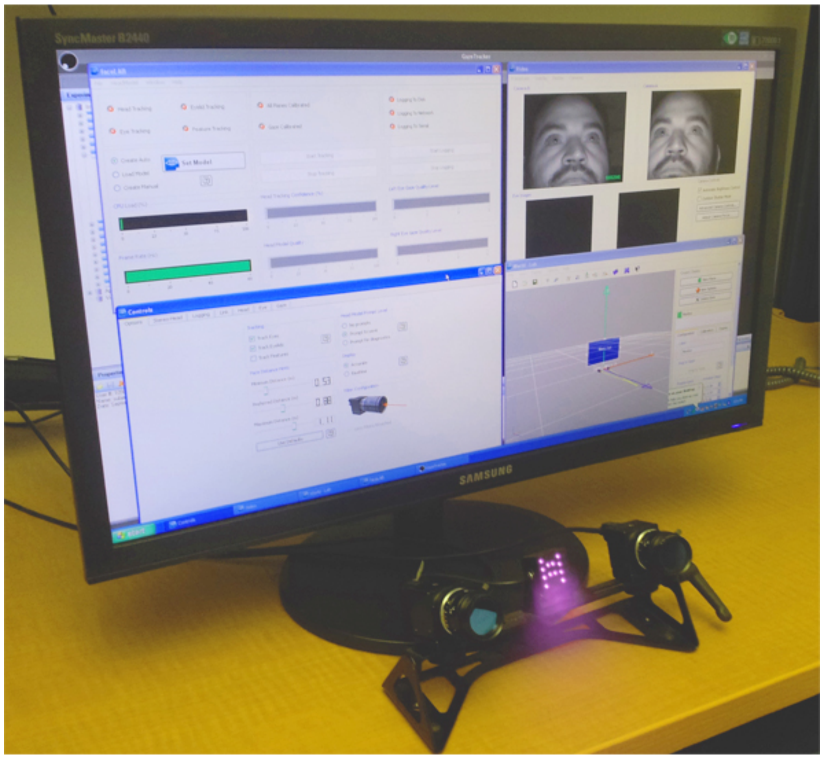


Figure 9.1 FaceLAB eye-tracker excerpt

perform the experiment including setting up eye-tracking system. The subjects were aware that they are going to perform RT tasks, but do not know the particular experiment research questions. There was only one case where we could not have fixations because of subject's eyeglasses. Thus, we excluded that subject and also one pilot subject from our study and analyse total 24 subjects.

9.2.4 Source Code Selection

We use a criteria to select the source code used in our eye-tracking experiment. We use Java programming language to perform our experiment because it is one of the OO languages that contains several different SCEs, *i.e.*, class, method, variable names, and comments. In addition, we select short and easy to read source code that could fit on one screen to have better control over eye-tracking system. However, the source code size is similar to previous eye-tracking studies (Sharif *et al.* (2012); Bednarik and Tukiainen (2006b)). We remove the automatically generated comments and we mix the domain and implementation related terms in the various SCEs, *e.g.*, class or method name. Font size was 20, and the 6 fragments of source code have 19, 18, 19, 18, 24, 28 LOCs.

9.2.5 Links, Tasks, and Questionnaires

We ask subjects to manually verify RT links. A subject can read a requirement and source code on screen to verify a RT link between them. If a subject thinks that *yes there must be a link*, s/he can simply write on the paper. We capture subjects' eye movement during the RT links verification task. There are six requirements and source codes that subjects must read to verify a link between them. We manually created RT links to evaluate the subjects' answers. To manually built the oracle or use pre-built oracles, we follow the same steps as described in Appendix A. One of the subjects performed a pilot-study to validate that the requirements used in the experiment are clear and simple and the source code on the screen is easy to read and understand. The tasks given to the subjects in our experiment consist of answering specific questions by viewing Java source code. Each question deals with the implementation of a requirement. For example, a Java code implements the requirement "CalculateArea class takes an input the radius of a circle to calculate its area". A subject can answer as 'true" if s/he thinks that the code is implementing the specified requirement. A replication package of all the questions and source code is available online[2].

9.2.6 Procedure

We conduct the experiment in a quiet, small room where the eye-tracking system is installed. We divide the experiment into four steps. In the first step, we provide a single page instruction and guidelines to the subjects to perform the experiment. In the second step, we ask the subjects to provide their Java programming experience in years, if they have any. Before running the experiment, we briefly give a tutorial to explain the procedure of the experiment and the eye-tracking system (e.g., how it works and what information is gathered by the eye-tracker).

The subjects are seated approximately $70cm$ away from the screen in a comfortable chair

2. http://www.ptidej.net/download/experiments/icsm12a/

Table 9.1 Questions of the Eye-Tracking Experiment).

ID	Question
1	This class takes as input the radius of a circle to calculate its area.
2	This class uses a JDBC driver to connect to a database and get data from myTable.
3	This class draws a circle on the JFrame.
4	This class uses the Java 2D API to draw lines and a rectangle in a JFrame.
5	This class compares two strings.
6	This class uses a JDBC driver to connect to a database and insert data in myTable.

with arms and head rests and the eye-tracker is calibrated. This process takes less than five minutes to complete. After this, the environment in front of them is just a Java source code's image with a question at the right side of the same image.

In the third step, we ask the subjects to read the source code, requirements, and verify traceability links between them. We instruct subjects that they can press space bar button when they find the answer. Space bar button will take them to a blank screen with next image number. Subjects could move easily to write down their answer on the paper. For each question, we ask subjects to spend adequate time to explore the code while we capture subjects' eye movements during traceability verification process.

In the last step, we ask subjects' feedback about source code readability and understandability. We also ask their source code entity preferences to verify RT links. For example, if they prefer more comments over variable names to verify a link. We ask this question to analyse if subjects followed the same pattern in eye-tracking experiment or not.

9.2.7 Analysis And Result

We perform the following analysis to answer **RQ1**, (see Section 9) and try to reject our null hypotheses. We have one independent variable: the SCEs and we use two dependent variables: (1) the total time of fixation spent by subjects on each SCE and (2) the percentage of correct answers. We use Taupe (De Smet *et al.* (2011)) to analyse the collected data. Taupe software system is developed in Ptidej research group[3] to help researchers visualise, analyse, and edit the data recorded by eye-tracking systems.

For each subject, we calculated the total fixation time that a subject spent at a specific SCE, *e.g.*, class or method name, in milliseconds. We apply Kruskal-Wallis rank sum test on four sets of SCEs to analyse if subjects have equal preference, in particular visual attention, for class, method, variable names and comments or they are all statistically different for them. Table 9.2 reports that p-value of Kruskal-Wallis test, for all the SCEs results, are statistically significant (the p-value is below than the standard significant value, *i.e.*, 0.05). In addition, a wide majority of the fixations that show subjects' visual attention are found on method names and comments. We analyse that as soon as the subjects read the requirement, they go directly to read method names and–or comments. Also, we analyse all the heatmaps consisting of cumulative fixations of all the subjects for RT verification task. In Figure 9.3, we also present the heatmap for one of our source codes which shows a large number of fixations on the method name, comments, and requirement.

In figure 9.2, X axis shows the number of subjects. Y axis shows the average time spent on each SCE and average correct answers. To show both average correct answers and time

3. http://www.ptidej.net/research/taupe/

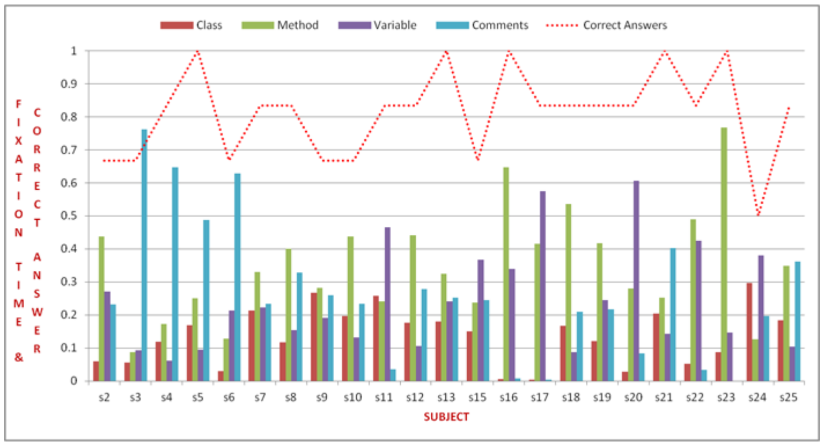
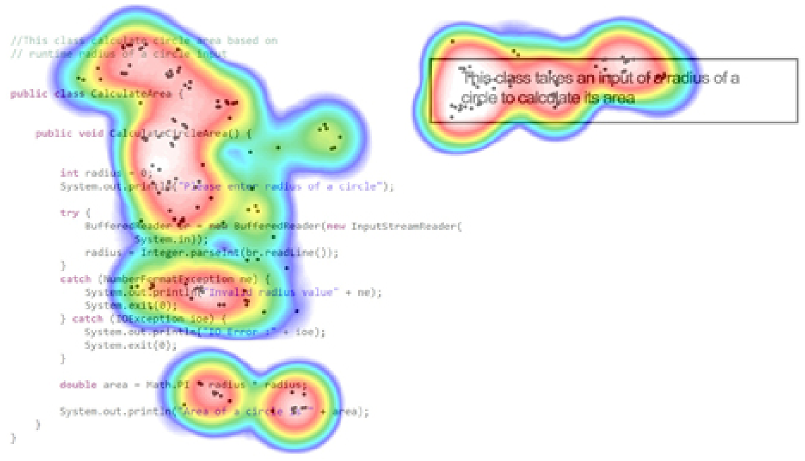


Figure 9.2 Distribution of Average Correct Answers and Average Fixations

spent on each source code entity, we normalise the total time. For example, subject S2 spent 6%, 44%, 27%, and 23% time on class, method, variable names, and comments respectively. Subject S2 on average gave 67% of correct answers. S1 was a pilot subject and for S14 we could not get enough fixation points due to eyeglasses. Thus, we excluded S1 and S14 from our study. The graph shows that high number of fixations at variables lead to less correct answers (or more wrong answer). All the subjects have different preference, in terms of importance, to comprehend the source code during RT task. Some subjects give more importance to method names and some comments. However, on average, subjects tent to look more at method names and comments.

Moreover, we observed that subjects' visual attention is more on SCEs that belong to domain concept than the implementation. For example, more fixations were at "radius" identifier and very less on "bufRead" identifier. We observed that subjects were more interested in domain related SCEs to comprehend the source code. We have on average 48% domain and 52% implementation SCEs in our experiment. Subjects on average spent 4865.3 milliseconds on domain identifiers, whereas only 1729.8 millisecond on implementation SCEs. Our post-experiment question about subjects' personal ranking for SCEs to comprehend the source code is also in agreement with observations we made with the eye-tracking.

Thus, we reject H_{01} and H_{02} and answer our **RQ1** that developers pay different visual attention on different SCEs to verify RT links. In particular, developers' visual attention changes as the SCEs change. In addition, developers almost doubles the visual attention if a SCE appears in domain or implementation concept of a program.



```
//This class calculate circle area based on
// runtime radius of a circle input

public class CalculateArea {

    public void CalculateCircleArea() {

        int radius = 0;
        System.out.println("Please enter radius of a circle");

        try {
            BufferedReader br = new BufferedReader(new InputStreamReader(
                    System.in));
            radius = Integer.parseInt(br.readLine());
        }
        catch (NumberFormatException ne) {
            System.out.println("Invalid radius value" + ne);
            System.exit(0);
        } catch (IOException ioe) {
            System.out.println("IO Error :" + ioe);
            System.exit(0);
        }

        double area = Math.PI * radius * radius;

        System.out.println("Area of a circle is " + area);
    }
}
```

This class takes an input of a radius of a circle to calculate its area

Figure 9.3 A heatmap showing the cumulative fixations of subjects. The colors red, orange, green and blue indicate the decrease in number of fixations from highest to lowest.

Table 9.2 Average time row shows average time spent on each SCE. Ranking row shows the ranking of each SCE based on its importance/fixations (4 means the most important).

	Class	**Method**	**Variable**	**Comments**	p-value
Avg. time (ms)	2317.25	5701.1	3181.81	4542.41	< 0.01
Ranking	1	4	2	3	–

9.3 Experiment Design: SE/IDF and DOI/IDF

This section describes our proposed weighting schemes SE/IDF and DOI/IDF to improve the accuracy of an IR technique by integrating developers' SCEs preferences. We tune the proposed weighting schemes based on the results that we achieved with our eye-tracking experiment, Section 9.1. We perform experiment on two datasets, *i.e.*, Pooka and iTrust, to analyse how much SE/IDF and DOI/IDF weighting schemes improve the F-measure. We explain all the steps of the experiment in this section. Proposed weighting schemes are supported by FacTrace (see Section 1.5).

IR techniques give importance to a term based on its probability or frequency in the corpus (see Section 2.1.2. We analyse that subjects have different preferences, for different SCEs, to perform RT verification task. Our proposed SE/IDF and DOI/IDF weighting schemes allow a developer to provide extra parameters to tune the weighting scheme to increase or decrease a term's importance in a corpus.In our proposed scheme, we consider each source code entity as a separate entity and assign weights according to its importance:

$$SC_j = C_j \cup M_j \cup V_j \cup Cmt_j$$

where, SC_j is the source code of a $class_j$ composed of entities that are class (C_j), method name (M_j), variable name (V_j), and comments (Cmt_j). A source code entity could belong to domain D_j or A_j implementation concept of a program.

$$scetf_{i,j} = \begin{cases} tf_{i,j} \times \alpha \ if \ t_i \in C_j \setminus R_j \\ tf_{i,j} \times (\alpha \times \lambda_1) \ if \ t_i \in C_j \cap R_j \\ tf_{i,j} \times \beta \ if \ t_i \in M_j \setminus R_j \\ tf_{i,j} \times (\beta \times \lambda_2) \ if \ t_i \in M_j \cap R_j \\ tf_{i,j} \times \gamma \ if \ t_i \in V_j \setminus R_j \\ tf_{i,j} \times (\gamma \times \lambda_3) \ if \ t_i \in V_j \cap R_j \\ tf_{i,j} \times \delta \ if \ t_i \in Cmt_j \setminus R_j \\ tf_{i,j} \times (\delta \times \lambda_4) \ if \ t_i \in Cmt_j \cap R_j \\ tf_{i,j} \times \Psi \ if \ R_j \setminus SC_j \end{cases}$$

Where scetf is **s**ource **c**ode **e**ntity **t**erm **f**requency, $\setminus$ not in set, $\alpha, \beta, \gamma, \delta$ are weights that can be tuned by a developer according to the importance of each SCE, R_j is a j^{th} requirement, and λ_k is term weight if it appears in both SCE and requirement. For example, if a same term appears in both class name and the requirement then it will multiply the weights by λ_1.

Where Ψ is the weight for a requirement term that does not appear in source code. In the case, where t_i appears in more than one of the subset $C_j \setminus R_j, M_j \setminus R_j, V_j \setminus R_j, Cmt_j \setminus R_j, C_j \cap R_j, M_j \cap R_j, V_j \cap R_j$, and $Cmt_j \cap R_j$, then we compute $scetf_{i,j}$ for each of the subset and we chose the maximum value. Thus, we define SE/IDF (source code entity/inverse document frequency) as follow:

$$SE/IDF_{i,j} = scetf_{i,j} \times IDF_i$$

Now, we extend our SE/IDF to consider the domain and implementation concepts of a program as follow:

$$DOITF_{i,j} = \begin{cases} tf_{i,j} \times \Upsilon \; if \; t_i \in D_j \\ tf_{i,j} \times \Phi \; if \; t_i \in A_j \end{cases}$$

Where $DOITF$ is **d**omain **o**r **i**mplementation **t**erm **f**requency, Υ and Φ are term weight. For example, if a term belongs to domain concept of a system then we multiply term by Υ. Thus, now we define DOI/IDF (domain or implementation/inverse document frequency) as follows:

$$DOI/IDF_{i,j} \; = \; (SE/IDF_{i,j} + DOITF_{i,j}) \; \times \; IDF_i$$

9.3.1 Objects

We use a criteria to select the source code that are used in our experiment (see Appendix A). We download the source code of Pooka v2.0 and iTrust v10.0 from their respective subversion repositories. The detailed descriptions and statistics of Pooka and iTrust are available in Appendix A.

9.3.2 Procedure

We perform the following steps to create RT links between source code and requirements using $LSI_{TF/IDF}$, $LSI_{SE/IDF}$, and $LSI_{DOI/IDF}$.

Gathering Requirements, Source Code and Building Oracles: We recover 90 functional requirements for Pooka in our previous work (Ali *et al.* (2011b)). We use these requirements to manually create traceability links between requirements and source code. In the case of Pooka, we follow the same steps as described in Appendix A. In the case of iTrust, we use online (see Appendix A) available requirements and manually built traceability links. $\text{Oracle}_{\text{iTrust}}$ and $\text{Oracle}_{\text{Pooka}}$ contain 546 and 183 traceability links respectively.

Extracting Concepts: We use LDA to extract domain and implementation concepts from the source code. LDA considers that documents are represented as a mixture of words acquired from different latent topics, where each topic is characterised by a distribution of words. More details on using LDA to extract domain concept could be found in (Maskeri *et al.* (2008)). LDA takes three parameters α, β, and k to create the topics as explained in Section 2.2.3. In this experiment, as in previous work (Abebe and Tonella (2011); Griffiths and Steyvers (2004)), the values for the parameters α and β are set to 25 (50/k) and 0.1 respectively. The value assigned for k is 2. We use MALLET[4] to extract concept from source code using LDA.

Generating Corpus: To process source code, we use Java parser (Ali *et al.* (2011a)) to extract all source code identifiers. The Java parser builds an AST of source code that can be queried to extract required identifiers, *e.g.*, class, method names, etc. Each Java source code file is thus divided in four SCEs and textual information is stored in their respective source code files. We use these files to apply proposed weighting schemes to create RT links.

Pre-processing the Corpus: We perform standard state-of-the-art (see Section 2.1.1) pre-processing steps. We remove non-alphabetical characters and then use the classic Camel Case and underscore algorithm to split identifiers into terms. Then, we perform the following steps to normalise requirements and SCEs: (1) convert all upper-case letters into lower-case and remove punctuation; (2) remove all stop words (such as articles, numbers, and so on); and (3) perform word stemming using the Porter Stemmer to bring back inflected forms to their morphemes.

Experiment Settings: Proposed weighting schemes require parameters to tune the equation. We assign weights to SE/IDF and DOI/IDF equations' parameters based on subjects' SCEs preferences during eye-tracking experiment. We use ranking based on fixations(see Table 9.2), observed during eye-tracking experiment, to define the weights of α, β, γ, and δ (see

4. http://mallet.cs.umass.edu

Table 9.3 Average Precision, Recall, and F-measure values and Wilcoxon p-values.

	Technique	Precision	Recall	F-Measure	p-value
Pooka	$LSI_{TF/IDF}$	14.71	21.07	9.52	< 0.01
	$LSI_{SE/IDF}$	18.30	24.86	12.47	
	$LSI_{DOI/IDF}$	25.73	26.42	14.56	
iTrust	$LSI_{TF/IDF}$	36.43	33.14	17.76	< 0.01
	$LSI_{SE/IDF}$	38.66	33.97	18.21	
	$LSI_{DOI/IDF}$	39.55	35.07	20.64	

Section 9.3). We normalise the ranking and assign weights to SCEs. To tune parameters of SE/IDF and DOI/IDF, we use the average fixation time that subjects spent on each SCE, domain, and implementation related SCEs. We assign $0.4, 0.3, 0.2$, and 0.1 (see Section 9.2.7) weight to β, δ, γ, and α respectively. If a term appears in both requirement and source code, then we simply double the weight of that term. We use lowest SCE weight and divide it by 2 to get the weight if a term only appears in requirement but not in source code. Thus, we assign weight 2 and 0.05 to λ_k and Ψ respectively. We observe that 74% and 26% of time subjects looked at domain and implementation identifiers respectively. Thus, we set 0.74 and 0.26 for Υ and Φ respectively.

RT Links Creation: We use LSI (Marcus and Maletic (2003c)) to create $LSI_{TF/IDF}$, $LSI_{SE/IDF}$, and $LSI_{DOI/IDF}$ RT links. The processed corpus is transformed into a term-by-document matrix, where each requirement and source code document is represented as a vector of terms. The values of the matrix cells represent the weights of the terms in the documents, which are computed using the traditional TF/IDF and proposed SE/IDF and DOI/IDF weighting schemes. Once all the requirements and source code documents have been represented in the LSI subspace, we compute the similarities between requirements and source code to create RT links. We take the cosine between their corresponding vector representations for calculating the similarity.

9.3.3 Analysis Method

We perform the following analysis on the recovered RT links to answer our research questions, RQ2 and RQ3, and attempt to reject our null hypotheses. We use $LSI_{SE/IDF}$, $LSI_{DOI/IDF}$, and $LSI_{TF/IDF}$ as independent variables and F-measure as a dependent variable to empirically attempt to reject the null hypotheses. We compute F-measure (Ali *et al.* (2011a)) of the $LSI_{SE/IDF}$, $LSI_{DOI/DIF}$, and $LSI_{TF/IDF}$ RT links in comparison to $Oracle_{Pooka}$ and $Oracle_{iTrust}$.

We use a threshold t to prune the set of traceability links, keeping only links whose similarity values are greater than 0. We use different values of t from 0.01 to 1 per step of 0.01 to obtain different sets of traceability links with varying precision and recall values. We use the same t number of threshold values, for comparing two techniques, to have the same number of data points for paired statistical test. We use these different sets to assess which technique provides better F-measure values. Then, we use the Wilcoxon rank sum test to assess whether the differences in F-measure values, in function of t, are statistically significant among the $LSI_{SE/IDF}$, $LSI_{DOI/DIF}$, and $LSI_{TF/IDF}$ techniques.

9.3.4 Results

Figure 9.4 shows the F-measure values of $LSI_{TF/IDF}$, $LSI_{SE/IDF}$, and $LSI_{DOI/IDF}$. It shows that $LSI_{DOI/IDF}$ provides better F-measures at all the different values of threshold t. Figure 9.4 shows that assigning different weights to SCEs, depending on their role and position in source code, provides better accuracy.

Table 9.3 shows that $LSI_{DOI/IDF}$ statistically improves on average up to 11.01%, 5.35%, 5.03% for precision, recall, and F-measure respectively. Results show that adding domain and implementation related SCEs information statistically improve over SE/IDF weighting. We perform Wilcoxon rank sum test on F-measure scores to analyse if the improvement is statistically significant or not.

We have statistically significant evidence to reject the H_{03} and H_{04} hypothesis for both datasets' results. Table 9.3 shows that the p-values are below than the standard significant value, *i.e.*, $\alpha = 0.05$. Thus, we answer the **RQ2** and **RQ3** as follow: integrating the developers knowledge, *i.e.*, SCEs preferences, in the IR technique statistically improves the accuracy. Adding the domain and implementation related SCEs information yield better accuracy than SE/IDF. Integrating developers' knowledge in the automated techniques could yield better accuracy, thus it is important to further observe and analyse developers to find out how they perform RT tasks.

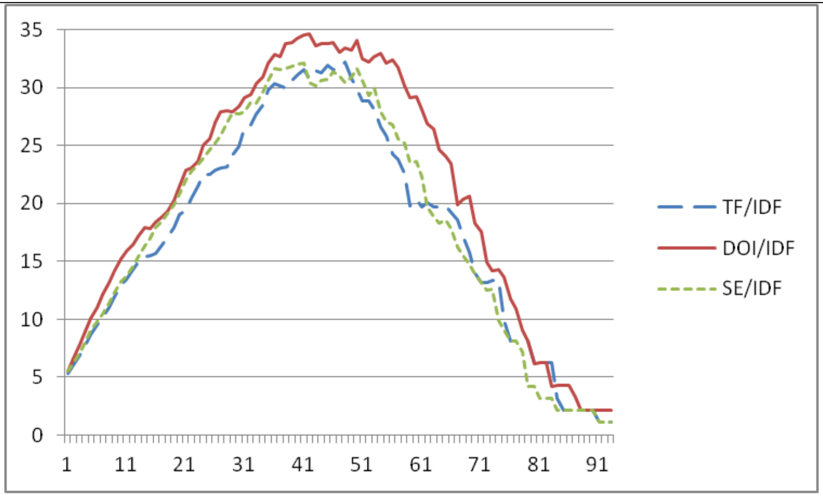
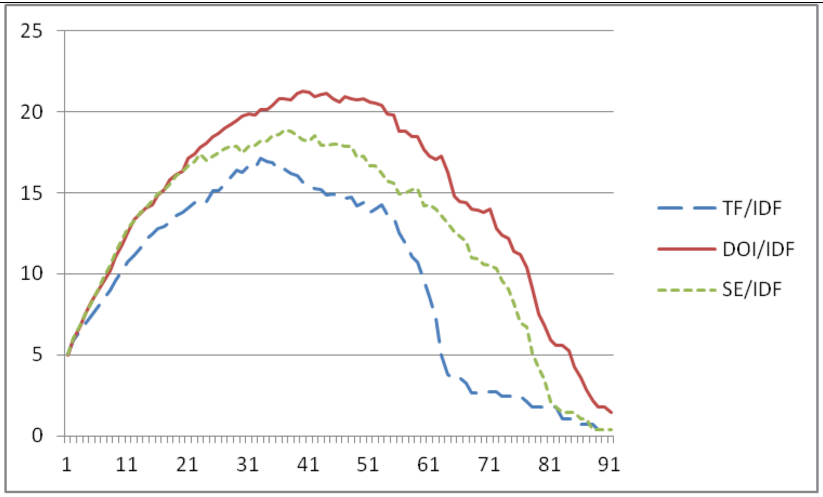
iTrust	Pooka
	

Figure 9.4 $LSI_{TF/IDF}$, $LSI_{SE/IDF}$, and $LSI_{DOI/IDF}$ F-measure values of iTrust and Pooka

9.4 Discussion

In response to the research questions defined in Section 9, we analyse that developers have different preferences for different SCEs. Mostly, developers read method names to search for the specific implementation of a requirement. We observe that as soon as developers read a requirement they start looking for specific methods of interest and read the comments. Developers least bother with the class names because one class may contain several functionalities (Ali *et al.* (2011a)). This is also the case for variable names because variable names could be object names of other classes. To avoid the bias of eye-tracking observations, we asked our subjects through a post-experiment questionnaire about the source code entities that help them to verify RT links. The results of cross verification questions are in agreement with the observations that we made with the eye-tracking system. This step gives us confidence in our findings and mitigates the threats to validity.

In regular document zoning, researchers mentioned that the title of a document is important (Kowalski (2010); Erol *et al.* (2006); Sun *et al.* (2004)). If we map a document structure into source code structure then class name is equal to a document title. In our experiment, we analyse that developers do not give preference to class name. We assigned more weight to class name and observed that it decreases the F-measure value. Thus, we conclude that a document is not same as source code. In addition, Figure 9.4 shows that if we only consider SCEs and assign weights, it does not improve as much as if we integrate the domain and implementation related SCEs information.

We integrated our eye-tracking observations in an IR-based technique, to analyse if it helps, to improve its accuracy when compared to $LSI_{TF/IDF}$. Our results show that observing how developers perform RT tasks and integrating those observations in an automated RT technique can improve its accuracy. This is the first step towards using eye-tracking systems to learn how developers perform RT tasks. Results are promising and exploring more into previously mentioned direction (observing developers) can improve the accuracy of automated techniques.

We analyse less effectiveness of SE/IDF and DOI/IDF on iTrust than on Pooka. We investigate the reason, we find that in iTrust, many classes do not contain all the SCEs, *i.e.*, class, method, variable name, and comments. In addition, we observe that iTrust contains more implementation related SCEs than domain. In the case of Pooka, we observe that it contains more domain related SCEs than implementation. We observe that if a dataset contains all the SCEs and more domain related identifiers then proposed SE/IDF and DOI/IDF weighting schemes perform better accuracy in terms of F-measure. However, even in the case of iTrust, it has less domain related SCEs and fewer classes containing all the SCEs; the proposed weighting schemes provide better accuracy. We observe that as the dataset size increases, SE/IDF and DOI/IDF provide much better results. For example, Pooka is ten times bigger than iTrust dataset and Pooka provides better results than iTrust. Our conjecture is that on bigger datasets SE/IDF and DOI/IDF would provide better results. However, we need more empirical studies to support this claim.

We set the weights of our weighting schemes based on the observations we made during our eye-tracking experiment. It is quite possible that using other weights may yield different results. However, with the current parameters, we observe that proposed weighting schemes outperform traditional TF/IDF. We will perform more case studies in future to analyse the impact of current static weight on other datasets.

There could be a limitation to our study and proposed weighting scheme is that we performed experiment on Java program at class level. Increasing the granularity at line level could not fit very well in our proposed model. Because then there would be only one SCE. However, our weighting equation, *i.e.*, DOI/IDF, still could be used at any level of granularity by only analysing domain and implementation level SCE.

We also made some interesting observations as follow:

9.4.1 Identifiers' Vocabulary

We observe that when a similar term appears in both requirement and source code, it helps developers to provide better accuracy than different ones. We suggest that developers must work closely with requirement engineers to understand the requirements and use the requirements' terms to name the SCE. Automated techniques are not 100% perfect yet and a developer must verify the recovered links that are provided by automated techniques and–or create the missing links. We observe that if the method names are not meaningful, then a developer may skip the whole method. However, it is quite possible that the method is implementing the requirement. We only observed this for method names because there was only one class per requirement. It is quite possible that, if there are meaningless class names, then a developer may skip the whole class as well without reading the whole source code. The identifiers' vocabulary does not make any threat to validity. Because the sometime the meaningless SCE was implementation related identifier that we were measuring during our eye-tracking experiment.

We observe that domain concept related SCE are focal points of developers during RT creation tasks. We recorded more fixations at domain concept identifiers. Using implementation identifiers may help developers to remember them and–or their data type etc. However, implementation identifiers do not help a lot in RT manual and automated tasks. We put more weight on application identifiers and observed decrease in F-measure value.

9.4.2 Male vs. Female

The main purpose of the study was not to compare the male and female subjects during RT recovery. However, we made interesting observations that may lead to future case studies on gender effect on RT tasks. There were 7 female subjects and 17 male subjects in our study. We observed that female subjects were more interested in details than males. For example, female subjects directly looked at the implementation of the source code without reading the comments or method names. They spent more time reading variable names and understanding them. Whereas, male subjects were more concerned about the high-level structure of the source code. They preferred more method names and comments. However, we cannot generalise this observation. More detailed studies are required to analyse gender difference on RT tasks.

9.5 Threats to Validity

Several threats limit the validity of our experiments. We now discuss these potential threats and how we control or mitigate them.

Construct validity: In our empirical study, they could be due to measurement errors. We use time spent on each AOI and percentages of correct answer to measure the subjects' performances. These measures are objective, even if small variations due to external factors, such as fatigue, could impact their values. We minimise this factor by using small source code and all the subjects finished the whole experiment within 20 minutes. In our empirical study, we use the widely-adopted metric F-measure to assess the IR technique as well as its improvement. The oracle used to evaluate the tracing accuracy could also impact our results. To mitigate such a threat, two Ph.D. students manually created traceability links for Pooka and then the third Ph.D. person verified the links. In addition, Pooka oracle was not specifically built for this empirical study. We used it in our previous studies (Ali *et al.* (2011b,a)). We use iTrust traceability oracle developed independently by the developers who did not know the goal of our empirical study.

Internal Validity: Learning threats do not affect our study for a specific experiment because we provide six different source codes and the subjects did not know the experiment goal. Selection of subject threats could impact our study due to the natural difference among the subjects' abilities. We analysed 24 out of 26 subjects with various experience to mitigate this threat. For our proposed technique, individual subjects' Java experience can cause some fluctuation in fixation that may lead to biased results. However, we minimised this threat by asking all subjects for their general source code comprehension preference at the end of the experiment. The results of our post-experiment questionnaire are in agreement with our eye-tracking experiment findings.

External Validity: The issue of whether students as subjects are representative of software professionals is one of the threats to generalise our result. However, our subjects are graduate students with the average 3.39 years of Java experience and they have good knowledge of Java programming. In addition, some of the subjects have industrial experience. We use small source code to perform the eye-tracking experiment and in some development environment, *e.g.*, Eclipse, developers' source code preferences could be different. Using development environment with eye-tracking system could weaken the control over the experiment and increase fixation offsets. Therefore, we use post experiment question to ask subjects about their general source code comprehension preference to avoid bias. Our empirical study is limited to two systems, Pooka and iTrust. They are not comparable to industrial projects, but the datasets used by other authors (Antoniol *et al.* (2002b); Marcus and Maletic (2003c); Poshyvanyk *et al.* (2007)) to compare different IR techniques have a comparable size. However, we cannot claim that we would achieve same results with other systems. Different systems with different identifiers' quality, reverse engineering techniques, requirements, using different software artifacts and other internal or external factors (Ali *et al.* (2012a)) may lead to different results. However, the two selected systems have different source code quality and requirements. Our choice reduces this threat to validity.

Conclusion validity: We pay attention not to violate assumptions of the performed statistical tests. In addition, we use non-parametric tests that do not make any assumption on the data distribution.

9.6 Summary

In this chapter, we conjecture that understanding how developers verify RT links could help improving the accuracy of IR-based techniques to recover RT links. To support this conjecture, we ask three research questions pertaining to (1) important SCEs used by developers to verify RT links; (2) domain vs. implementation-related entities; and (3) the use of SCEs to propose the new weighting schemes, SE/IDF and DOI/IDF, and to build a LSI-based technique to recover RT links.

We answer these RQs as follows: first, we analysed the eye movements of 24 subjects to identify their preferred SCEs when they read source code to verify RT links using an eye-tracker. Results show that subjects have different preferences for class names, method names, variable names, and comments. They search/comprehend source code by reading method names and comments mostly. We reported that, as soon as developers read a requirement, they pay immediately attention to method names and then comments. We analysed that subjects gives more importance to a SCE if it appears in domain concept of a program.

Second, we propose two new weighting schemes namely SE/IDF and DOI/IDF, to assign different weights to each SCE. For DOI/IDF, we need domain and implementation SCEs for each class. In this chapter, we use LDA to distinguish domain concepts from implementation. More advance and complex techniques (Abebe and Tonella (2011)) could be used to separate domain and implementation concepts.

Third, using developers' preferred SCEs and their belonging to either domain or implementation, we then propose SE/IDF and DOI/IDF, new weighting schemes based on the results of the two previous RQs. SE/IDF and DOI/IDF replace the usual TF/IDF weighting scheme so that we could integrate the observations about the subjects' preferred SCEs ranked list into TF/IDF. We use a LSI that uses SE/IDF and DOI/IDF, on two datasets, iTrust and Pooka to show that, in general, the proposed schemes have a better accuracy than TF/IDF in terms of F-measure. In both systems, proposed weighting schemes statistically improve the accuracy of the IR-based technique. We analyse that assigning more weight to domain concepts yields better accuracy.

Part IV

Conclusion and Future Work

CHAPTER 10

CONCLUSION

Requirements traceability is an important part of software development and maintenance. Manually creating traceability links is a laborious and effort-consuming task. Many researchers (see Section 3) have proposed automated and semi-automated techniques to create links among requirements and software artifacts. Among the proposed techniques, IR techniques (Abadi *et al.* (2008); Antoniol *et al.* (2002b); Marcus and Maletic (2003a)) have proven to be efficient in creating links among requirements and software artifacts. However, some factors (Ali *et al.* (2012a)) impact the input of RTAs, consequently the accuracy of RTAs. Ali *et al.* (2012a) suggested that controlling/avoiding the impact of these factors could yeild RTAs with better results. All the techniques proposed in literature (Binkley *et al.* (2009); Enslen *et al.* (2009); Gleich *et al.* (2010); Guerrouj *et al.* (2011)), to improve the quality of RTAs' input, are not fully automated. For example, automated identifiers' splitting techniques (Enslen *et al.* (2009); Guerrouj *et al.* (2011)) are not completely automated and yet perfect. To perfectly split an identifier that does not follow any naming convention, *e.g.*, CamelCase or under_score, a developer must manually split them (Dit *et al.* (2011b)). However, including dynamic information (Dit *et al.* (2011b); Poshyvanyk *et al.* (2007)) could mitigate the impact of imperfect splitting techniques.

Thus, in this dissertation, our thesis :

Adding more sources of information and combining them with IR techniques could improve the accuracy, in terms of precision and recall, of IR techniques for requirements traceability.

To prove our hypothesis, we proposed to consider more sources of information as experts and a novel trust-model to combine their opinions.

10.0.1 Creation of Experts

We proposed four approaches, *i.e.*, Histrace, BCRTrace, Partrace, and developer's knowledge, to create experts. All these four approaches are independent and could be replaced with each other in different scenarios. Histrace uses software repositories to create experts. For example, Histrace links requirements to CVS/SVN commits to create an expert $Histrace_{commits}$. If software repositories are not available for a software system then BCRTrace or Partrace are useful to create experts. BCRTrace uses BCRs among classes to create experts. In this dissertation, we consider four BCRs, *i.e.*, aggregation, association, inheritance, and use. It is quite possible developers do not use all these four BCRs to implement a feature. In this situation, Partrace can create experts. Partrace only depends on source code structure. It partitions source code into four parts, *i.e.*, class name, method name, variable name, and comments, and use each part as an expert.

10.0.2 Combining Experts' Opinion

Each expert has its own trust on a baseline link. More the experts vote a link with high trust, more we could trust the link. To reevaluate the trust worthiness of a link, we must combine the opinions of all the experts. We propose a trust-model Trumo, inspired by Web trust-models of users (Berg and Van (2001); Palmer *et al.* (2000); Koufaris and Hampton-Sosa (2004); Artza and Gil (2007); Grandison and Sloman (2000)). Trumo take two inputs, *i.e.*, baseline links recovered by an IR technique and links recovered by experts. Trumo discards a link if no experts vote for the link. For the remaining links, Trumo counts how many experts voted for a link and the trust-value of each expert. Trumo uses a weighting scheme to assign weight to each expert and the total number of times experts voted for a link.

10.0.3 Usage of Experts and their Opinions

We proposed four approaches to utilise the opinion of these experts on the baseline links recovered by an IR technique. We experimentally show that we can use these experts and the trust-model to create RT links with better precision and recall.

Trustrace: Trustrace uses Histrace to create experts, Trumo to combine their opinions, and DynWing to assign weights to each expert. We apply Trustrace on four systems, *i.e.*, jEdit, Pooka, Rhino, and SIP Communicator, to compare the accuracy of its traceability links with those recovered using state-of-the-art IR techniques, based on the VSM and JSM, in terms of precision and recall. The results of Trustrace have up-to 22.7% more precision and 7.66% more recall than those of the other techniques, on average.

LIBCROOS: We proposed LIBCROOS, which uses BCRTrace to create experts, which vote on the baseline links recovered by an IR technique. We customised Trumo model to combine BCRs with IR techniques to link bug reports to source code. We measured the accuracy of LIBCROOS and IR techniques by rank of culprit class in the ranked list generated by IR technique or LIBCROOS. An empirical study on four systems, *i.e.*, Jabref, Lucene, muCommander, and Rhino, to compare the accuracy of LIBCROOS with two IR techniques, *i.e.*, LSI and VSM, showed that LIBCROOS improves the accuracy of both IR techniques statistically when compared to LSI and VSM alone.

COPARVO: We proposed COPARVO, which uses Partrace to create experts. COPARVO considers each partition, *i.e.*, class names, method name, variable name, and comments, created by Partace as an extra source of information to verify the links recovered by an IR technique. We applied COPARVO on three systems, *i.e.*, Pooka, SIP Communicator, and iTrust, to filter out false positive links recovered via the information retrieval approach *i.e.*, VSM without decreasing the precision and recall. The results show that COPARVO significantly improves the accuracy of recovered RT links up to 8.11% precision and 2.67% recall. COPARVO was also able to reduce 83% of manual effort required to manually remove false positive links.

An Improved Weighting Scheme: We use developers' knowledge as extra source of information. We gathered developers' knowledge using an eye-tracker. We observed that developers have different preferences for different source code parts. We combined developers' knowledge with existing TF/IDF, to propose a new weighting scheme, *i.e.*, SE/IDF (source code entity/inverse document frequency) and DOI/IDF (domain or implementation/inverse document frequency). We applied propose weighting scheme on Pooka and iTrust. The results showed that proposed weighting schemes can improve the accuracy of IR technique, *i.e.*, LSI, up to 11% better precision and 5.35% better recall.

Summary: This dissertation analysed the benefits of using more sources of information for traceability link recovery between software artifacts. We proposed four approaches, *i.e.*, Histrace, BCRTrace, Partrace, and developer's knowledge, to create the experts. All the proposed techniques effectively created experts, which could vote on the baseline links recovered by an IR technique or which could be integrated in IR techniques, *i.e.*, developer's knowledge. These techniques could be used in different scenarios based on available more sources of information. We propose a trust-model to combine these experts and their opinions. Trumo discards and re-rank links based on the experts' opinions. We proposed four approaches, Trustrace, LIBCROOS, COPARVO, and improved term weighting, to utilise all these experts and their opinions. We showed that all the proposed approaches improved the accuracy of existing IR techniques. The results proved our thesis that adding more sources of information help to improve the accuracy of existing IR techniques. We found that adding external, *i.e.*, software repositories, source of information, not directly related to source code *e.g.*, source code parts and BCRs, provides better results. We also analysed that developers do not (i) write enough information for bug reports and–or CVS/SVN logs (ii) properly comment source code and use appropriate identifiers (iii) use more BCRs while implementing a feature (iv)use documentation vocabulary while writing source code. All previously mentioned points directly impact the accuracy of IR techniques.

10.1 Limitations

Despite the above promising results, our thesis is threatened by following:

Limitation of Trustrace: Trustrace depends on software repositories and the quality of textual information available in them. However, for some small, in-house projects, and–or in some other cases, developers may not use software repositories to track software evolution. In this case, Trustrace could not be fully applied. We observed in Chapter 6 that only 4% of *SIP* SVN commits were linked to bug reports. Therefore, we did not find much evidence for many links in SVN commits and this lack of evidence yielded many links from the baseline set to be removed, following our constraints in Equation 5.1, and, consequently, a lower recall than IR techniques. We also observed that the conceptual distance between software repositories and requirements was quite large. Figure 6.5 shows that in the case of *Pooka* and *SIP* both JSM and VSM return low similarity values. Low similarity values among different documents would result into low precision and recall values for the traceability links (Ali *et al.* (2012a)). Thus, if the quality of software repositories' data is poor then Trustrace may not provide promising results.

Limitation of LIBCROOS: LIBCROOS is based on the usage of BCRs as experts to vote on recovered links by an IR technique. We observed during our experiments in Chapter 7 that sometimes developers do not respect OO rules. They do not use any BCR to implement a single feature. Thus, in the absence of BCRs, LIBCROOS may not perform very well. However, LIBCROOS has a generic model so other OO metrics, *e.g.*, coupling and cohesion, could be analysed to put culprit classes at the top. We observed that in most of the experiments, inheritance is an important BCR to improve the accuracy of IR techniques. This observation raises a question that LIBCROOS may not perform well on procedural programming languages, *e.g.*, C and Fortran, and–or other BCR could be more important. We manually selected λ and it is quite possible that using different λ values, we may not have similar results.

Limitation of COPARVO: COPARVO uses Partrace to divide source code into partitions and each partition votes on baseline links recovered by an IR technique. COPARVO's current model only allows traceability from requirements to source code. However, we cannot claim that it will produce the same results on other pairs of software artifacts, or on traceability of requirements to requirements or UML diagrams to use cases. We only analysed three systems, *i.e.*, iTrust, Pooka, and SIP Communicator and one IR technique, *i.e.*, VSM. It is possible on other software systems or IR technique; we may not obtain the same results.

Limitation of SE/DOI: We only performed the experiment with students and not with the real developers. It is possible that real developers have different preferences for SCEs. In addition, to have better control over the eye-tracker, we only used small source code fragments. Mostly, there was only one method per class. We cannot claim that on large source code fragments developers will have the same SCEs' preferences. We tuned our proposed weighting schemes, *i.e.*, SE/IDF and DOI/IDF, based on the experiment results achieved with the eye-tracker experiment. However, it is possible that using other parameters to tune our weighting scheme could yield different results.

10.2 Future Work

In this dissertation, we have verified our thesis and showed that considering more sources of information to recover traceability links among requirements and source code, improves the accuracy of IR-based RTAs. Future work would be devoted to further experiments with proposed approaches and assess FacTrace (see Section 1.5) on larger software system, in particular industrial partners interested in using FacTrace. We will also fill the missing combinations in the Table 5.1. Below, we describe how we plan to extend the work presented in this dissertation:

Trustrace: We observed that Trustrace improves the accuracy of IR techniques, despite the above mentioned limitations. In the future, we want to perform more experiments to generalise the results. We plan to implement more instances of Histrace, in particular using e-mails and threads of discussions in forums. We will use our Trumo model in other software engineering fields, in particular, test-case prioritisation using various prioritisation approaches, anti-pattern detection using user feedback, and concept location using execution traces. We will deploy Trustrace in a development environment and perform experiments with real developers to analyse how effectively Trustrace can help developers in recovering traceability links. We also plan to use advanced matching techniques between bug reports and CVS/SVN commit messages (Bachmann *et al.* (2010); Wu *et al.* (2011)).

LIBCROOS: Despite the above mentioned limitations of LIBCROOS, we observed that LIBCROOS improves the accuracy of IR techniques even with a single BCR. We are planning to extend LIBCROOS as follows: first, we will consider more BCRs and other OO metrics, *e.g.*, coupling and cohesion, to analyse the impact of different relationships on bug location. Second, we will apply our approach on different software comprehension and maintenance activities. Third, we will analyse more datasets and programming languages, *e.g.*, C and Fortran, to increase the generalisability of our findings. We will also perform an in-depth analysis of the λ (see Section 7.1.4) value effect on other datasets. Lastly, we will use other

IR techniques, *e.g.*, JSM, to quantify the improvement using our proposed approach.

COPARVO: First, we are planning to apply COPARVO on heterogeneous software artifacts to analyse accuracy improvement and effort reduction. Second, we will add more datasets to generalise our findings. Lastly, we will use other IR techniques to quantify the improvement using our proposed approach. Our empirical case studies demonstrated that each source code partition has potential but not all the partitions are equal in recovering traceability links. We will use different weighting schemes for each source code partition to measure the accuracy of IR-based approaches.

SE/IDF + DOI/IDF: There are several ways in which we are planning to continue this work. First, we will apply SE/IDF and DOI/IDF on more datasets. Second, we will analyse in which source code entity (SCE) a requirement term plays a more important role. Third, we will apply the proposed weighting schemes on heterogeneous software artifacts to analyse the improvement of the accuracy. Fourth, we will apply SE/IDF and DOI/IDF on feature location techniques. Lastly, we will use some automated techniques to tune SE/IDF and DOI/IDF parameters to improve their accuracy.

COPARVO + (SE/IDF + DOI/IDF): We analysed that COPARVO was able to significantly reduce the manual effort to verify false positive links recovered by an IR techniques. However, the accuracy of the IR technique was not greatly improved. On the other hand, we observed that the proposed weighting scheme, *i.e.*, $SE/IDF + DOI/IDF$, greatly improved the accuracy of IR techniques. The main conjecture behind both approaches was to treat each SCE differently. Thus, we are planning to integrate COPARVO and the proposed weighting schemes, *i.e.*, $SE/IDF + DOI/IDF$, to assign weights to each SCE accordingly and each SCE would be able to vote on the recovered links by an IR technique.

REFERENCES

ABADI, A., NISENSON, M. and SIMIONOVICI, Y. (2008). A traceability technique for specifications. *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on*. 103 –112.

ABEBE, S. and TONELLA, P. (2011). Towards the extraction of domain concepts from the identifiers. *18th Working Conference on Reverse Engineering (WCRE)*. 77 –86.

ALI, N., GUÉHÉNEUC, Y.-G. and ANTONIOL, G. (2011a). Requirements traceability for object oriented systems by partitioning source code. *Reverse Engineering (WCRE), 2011 18th Working Conference on*. 45 –54.

ALI, N., GUÉHÉNEUC, Y.-G. and ANTONIOL, G. (2011b). Trust-based requirements traceability. S. E. Sim and F. Ricca, éditeurs, *Proceedings of the* 19th *International Conference on Program Comprehension*. IEEE Computer Society Press. 10 pages.

ALI, N., GUÉHÉNEUC, Y.-G. and ANTONIOL, G. (2012a). *Factors Impacting the Inputs of Traceability Recovery Approaches*. Springer-Verlag, New York.

ALI, N., GUÉHÉNEUC, Y.-G. and ANTONIOL, G. (2012b). Trustrace: Mining software repositories to improve the accuracy of requirement traceability links. *IEEE Transactions on Software Engineering*, 99. To appear.

ALI, N., SABANÉ, A., GUÉHÉNEUC, Y.-G. and ANTONIOL, G. (2012c). Improving bug location using binary class relationships. M. Ceccato and Z. Li, éditeurs, *Proceedings of the* 12th *International Working Conference on Source Code Analysis and Manipulation*. IEEE Computer Society Press. 10 pages.

ALI, N., SHARAFI, Z., GUÉHÉNEUC, Y.-G. and ANTONIOL, G. (2012d). An empirical study on requirements traceability using eye-tracking. M. D. Penta and J. I. Maletic, éditeurs, *Proceedings of the* 28th *International Conference on Software Maintenance (ICSM)*. IEEE Computer Society Press. 10 pages.

ALI, N., WU, W., ANTONIOL, G., PENTA, M. D., GUÉHÉNEUC, Y.-G. and HAYES, J. H. (2010). A novel process and its implementation for the multi-objective miniaturization of software. Rapport technique EPM-RT-2010-04, Ecole Polytechnique de Montreal. Technical Report.

ALI, N., WU, W., ANTONIOL, G., PENTA, M. D., GUÉHÉNEUC, Y.-G. and HAYES, J. H. (2011c). Moms: Multi-objective miniaturization of software. *27th IEEE International Conference on Software Maintenance*. IEEE, IEEE CS Press, 10.

ALLEN, E. (2002). *Bug Patterns in Java.* APress L. P.

ANTONIOL, G., CANFORA, G., CASAZZA, G., DE LUCIA, A. and MERLO, E. (2002a). Recovering traceability links between code and documentation. *IEEE Trans. Softw. Eng.*, 28, 970–983.

ANTONIOL, G., CANFORA, G., CASAZZA, G., LUCIA, A. D. and MERLO, E. (2002b). Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering*, 28, 970–983.

ANTONIOL, G., CAPRILE, B., POTRICH, A. and TONELLA, P. (2000a). Design-code traceability for object-oriented systems. *Annals of Software Engineering*, 9, 35–58.

ANTONIOL, G., CAPRILE, B., POTRICH, A. and TONELLA, P. (2001). Design-code traceability recovery: selecting the basic linkage properties. *Science of Computer Programming*, 40, 213–234.

ANTONIOL, G., CASAZZA, C. and CIMITILE, A. (2000b). Traceability recovery by modeling programmer behavior. *Reverse Engineering, 2000. Proceedings. Seventh Working Conference on.* IEEE, 240–247.

ARTZA, D. and GIL, Y. (2007). A survey of trust in computer science and the semantic web. *Web Semantics: Science, Services and Agents on the World Wide Web*, 5, 58 – 71.

ASUNCION, H., ASUNCION, A. and TAYLOR, R. (2010). Software traceability with topic modeling. *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1.* ACM, New York, NY, USA, 95–104.

BACHMANN, A., BIRD, C., RAHMAN, F., DEVANBU, P. and BERNSTEIN, A. (2010). The missing links: bugs and bug-fix commits. *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering.* ACM, New York, NY, USA, FSE '10, 97–106.

BAEZA-YATES, R. and RIBEIRO-NETO, B. (1999a). *Modern Information Retrieval.* Addison-Wesley.

BAEZA-YATES, R. A. and RIBEIRO-NETO, B. (1999b). *Modern Information Retrieval.* Addison-Wesley, New York.

BEDNARIK, R. and TUKIAINEN, M. (2006a). An eye-tracking methodology for characterizing program comprehension processes. *Proceedings of the 2006 symposium on Eye tracking research & applications.* ACM, New York, NY, USA, ETRA '06, 125–132.

BEDNARIK, R. and TUKIAINEN, M. (2006b). An eye-tracking methodology for characterizing program comprehension processes. *Proceedings of the 2006 symposium on Eye tracking research & applications.* ACM, New York, NY, USA, 125–132.

BERG, R. and VAN, J. M. L. (2001). Finding symbolons for cyberspace: addressing the issues of trust in electronic commerce. *Production Planning and Control*, 12, 514–524(11).

BIGGERS, L., BOCOVICH, C., CAPSHAW, R., EDDY, B., ETZKORN, L. and KRAFT, N. (2012). Configuring latent dirichlet allocation based feature location. *Empirical Software Engineering*, 1–36.

BINKLEY, D., FEILD, H., LAWRIE, D. and PIGHIN, M. (2009). Increasing diversity: Natural language measures for software fault prediction. *Journal of Systems and Software*, 82, 1793–1803.

BLEI, D. M., NG, A. Y. and JORDAN, M. I. (2003). Latent dirichlet allocation. *J. Mach. Learn. Res.*, 3, 993–1022.

BOOCH, G. (1991). *Object-Oriented Design with Applications*. Benjamin/Cummings Publishing Company.

BROOKS, R. (1983). Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 18, 543–554.

BUNGE, M. (1977). *Treatise on Basic Philosophy: Vol. 3: Ontology I: The Furniture of the World*. Reidel, Boston MA.

CAPOBIANCO, G., DE LUCIA, A., OLIVETO, R., PANICHELLA, A. and PANICHELLA, S. (2009). On the role of the nouns in ir-based traceability recovery. *Program Comprehension, 2009. ICPC'09. IEEE 17th International Conference on*. IEEE, 148–157.

DAGENAIS, B., OSSHER, H., BELLAMY, R. K. E., ROBILLARD, M. P. and DE VRIES, J. P. (2010). Moving into a new software project landscape. *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*. ACM, New York, NY, USA, ICSE '10, 275–284.

DAY, W. and EDELSBRUNNER, H. (1984). Efficient algorithms for agglomerative hierarchical clustering methods. *Journal of classification*, 1, 7–24.

DE LUCIA, A., DI PENTA, M., OLIVETO, R., PANICHELLA, A. and PANICHELLA, S. (2011). Improving ir-based traceability recovery using smoothing filters. *Program Comprehension (ICPC), 2011 IEEE 19th International Conference on*. 21 –30.

DE LUCIA, A., DI PENTA, M., OLIVETO, R. and ZUROLO, F. (2006). Improving comprehensibility of source code via traceability information: a controlled experiment. *Program Comprehension, 2006. ICPC 2006. 14th IEEE International Conference on*. 317 –326.

DE LUCIA, A., FASANO, F., OLIVETO, R. and TORTORA, G. (2004). Enhancing an artefact management system with traceability recovery features. *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*. IEEE, 306–315.

DE SMET, B., LEMPEREUR, L., SHARAFI, Z., GUÉHÉNEUC, Y.-G., ANTONIOL, G. and HABRA, N. (2011). Taupe: Visualising and analysing eye-tracking data. *System for Science of Computer Programming.*

DEERWESTER, S., DUMAIS, S., FURNAS, G., LANDAUER, T. and HARSHMAN, R. (1990). Indexing by latent semantic analysis. *Journal of the American society for information science*, 41, 391–407.

DIT, B., GUERROUJ, L., POSHYVANYK, D. and ANTONIOL, G. (2011a). Can better identifier splitting techniques help feature location? *Program Comprehension (ICPC), 2011 IEEE 19th International Conference on.* 11 –20.

DIT, B., REVELLE, M., GETHERS, M. and POSHYVANYK, D. (2011b). Feature location in source code: a taxonomy and survey. *Journal of Software Maintenance and Evolution: Research and Practice.*

DÖMGES, R. and POHL, K. (1998). Adapting traceability environments to project-specific needs. *Communications of the ACM*, 41, 54–62.

DUCHOWSKI, A. (2002). A breadth-first survey of eye-tracking applications. *Behavior Research Methods*, 34, 455–470.

DUCHOWSKI, A. (2007). *Eye tracking methodology: Theory and practice.* Springer-Verlag New York Inc.

EADDY, M., AHO, A., ANTONIOL, G. and GUÉHÉNEUC, Y.-G. (2008a). Cerberus: Tracing requirements to source code using information retrieval dynamic analysis and program analysis. *ICPC '08: Proceedings of the 2008 The 16th IEEE International Conference on Program Comprehension.* IEEE Computer Society, Washington DC USA, 53–62.

EADDY, M., AHO, A., ANTONIOL, G. *ET AL.* (2008b). Cerberus: Tracing requirements to source code using information retrieval, dynamic analysis, and program analysis. *The 16th IEEE International Conference on Program Comprehension.* IEEE, 53–62.

EADDY, M., ZIMMERMANN, T., SHERWOOD, K. D., GARG, V., MURPHY, G. C., NAGAPPAN, N. and AHO, A. V. (2008c). Do crosscutting concerns cause defects? *IEEE Transactions on Software Engineering*, 34, 497–515.

ENSLEN, E., HILL, E., POLLOCK, L. and VIJAY-SHANKER, K. (2009). Mining source code to automatically split identifiers for software analysis. *Mining Software Repositories, 2009. MSR'09. 6th IEEE International Working Conference on.* IEEE, 71–80.

EROL, B., BERKNER, K. and JOSHI, S. (2006). Multimedia thumbnails for documents. *Proceedings of the 14th annual ACM international conference on Multimedia.* ACM, New York, NY, USA, MULTIMEDIA '06, 231–240.

FRAKES, W. B. and BAEZA-YATES, R. (1992). *Information Retrieval: Data Structures and Algorithms*. Prentice-Hall, Englewood Cliffs, NJ.

GETHERS, M., OLIVETO, R., POSHYVANYK, D. and LUCIA, A. D. (2011). On integrating orthogonal information retrieval methods to improve traceability recovery. *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*. 133 –142.

GLEICH, B., CREIGHTON, O. and KOF, L. (2010). Ambiguity Detection: Towards a Tool Explaining Ambiguity Sources. *Requirements Engineering: Foundation for Software Quality*, 218–232.

GOTEL, O. C. Z. and FINKELSTEIN, C. W. (1994). An analysis of the requirements traceability problem. *Requirements Engineering., Proceedings of the First International Conference on*, 94–101.

GRANDISON, T. and SLOMAN, M. (2000). A survey of trust in internet applications. *Communications Surveys Tutorials, IEEE*, 3, 2 –16.

GRECHANIK, M., MCKINLEY, K. and PERRY, D. (2007). Recovering and using use-case-diagram-to-source-code traceability links. *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM, 95–104.

GRIFFITHS, T. and STEYVERS, M. (2004). Finding scientific topics. *Proceedings of the National Academy of Sciences of the United States of America*, 101, 5228–5235.

GUÉHÉNEUC, Y.-G. (2005). Ptidej: Promoting patterns with patterns. *Proceedings of the 1st ECOOP workshop on Building a System using Patterns*. Springer-Verlag.

GUÉHÉNEUC, Y.-G. and ALBIN-AMIOT, H. (2004). Recovering binary class relationships: putting icing on the uml cake. *SIGPLAN Not.*, 39, 301–314.

GUERROUJ, L., DI PENTA, M., ANTONIOL, G. and GUÉHÉNEUC, Y. (2011). Tidier: an identifier splitting approach using speech recognition techniques. *Journal of Software Maintenance and Evolution: Research and Practice*.

HAYES, J. H., ANTONIOL, G. and GUÉHÉNEUC, Y.-G. (2008). Prereqir: Recovering pre-requirements via cluster analysis. *Reverse Engineering, 2008. WCRE '08. 15th Working Conference on*. 165 –174.

HAYES, J. H., DEKHTYAR, A., SUNDARAM, S. K., HOLBROOK, E. A., VADLAMUDI, S. and APRIL, A. (2007). Requirements tracing on target (retro): improving software maintenance through traceability recovery. *ISSE*, 3, 193–202.

HAYES, J. H., DEKHTYAR, A., SUNDARAM, S. K. and HOWARD, S. (2004). Helping analysts trace requirements: An objective look. *RE '04: Proceedings of the Requirements*

Engineering Conference, 12th IEEE International. IEEE Computer Society, Washington, DC, USA, 249–259.

HOFMANN, T. (1999). Probabilistic latent semantic indexing. *Proceedings of the 22nd annual international ACM SIGIR conference on Research and development in information retrieval.* ACM, 50–57.

HOFMANN, T. (2001). Unsupervised learning by probabilistic latent semantic analysis. *Machine Learning*, 42, 177–196.

JACKSON, D. and WAINGOLD, A. (1999). Lightweight extraction of object models from bytecode. *Proceedings of the 21st international conference on Software engineering.* New York, NY, USA, 194–202.

JONES, K. (1972). A statistical interpretation of term specificity and its application in retrieval. *Journal of documentation*, 28, 11–21.

KAGDI, H. and MALETIC, J. (2007). Software repositories: A source for traceability links. *International Workshop on Traceability in Emerging Forms of Software Engineering.* 32 –39.

KAGDI, H., MALETIC, J. and SHARIF, B. (2007). Mining software repositories for traceability links. *Program Comprehension, 2007. ICPC '07. 15th IEEE International Conference on.* 145 –154.

KAGDI, H., YUSUF, S. and MALETIC, J. I. (2006). Mining sequences of changed-files from version histories. *Proceedings of the 2006 international workshop on Mining software repositories.* New York, NY, USA, MSR '06, 47–53.

KOO, S. R., SEONG, P. H., YOO, J., CHA, S. D. and YOO, Y. J. (2005). An effective technique for the software requirements analysis of npp safety-critical systems, based on software inspection, requirements traceability, and formal specification. *Reliability Engineering & System Safety*, 89, 248 – 260.

KOUFARIS, M. and HAMPTON-SOSA, W. (2004). The development of initial trust in an online company by new customers. *Information & Management*, 41, 377 – 397.

KOWALSKI, G. (2010). *Information retrieval architecture and algorithms.* Springer-Verlag New York Inc.

LEFFINGWELL, D. (1997). Calculating your return on investment from more effective requirements management. *Available from Rational, URL http://www. rational. com/media/whitepapers/roi1. pdf.*

LIU, D., MARCUS, A., POSHYVANYK, D. and RAJLICH, V. (2007). Feature location via information retrieval based filtering of a single scenario execution trace. *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering.* ACM, 234–243.

LUCIA, A. D., FASANO, F., OLIVETO, R. and TORTORA, G. (2007). Recovering traceability links in software artifact management systems using information retrieval methods. *ACM Trans. Softw. Eng. Methodol.*, 16.

LUKINS, S., KRAFT, N. and ETZKORN, L. (2008). Source code retrieval for bug localization using latent dirichlet allocation. *Reverse Engineering, 2008. WCRE '08. 15th Working Conference on*. 155 –164.

MADER, P., GOTEL, O. and PHILIPPOW, I. (2008). Enabling automated traceability maintenance by recognizing development activities applied to models. *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, Washington, DC, USA, 49–58.

MALETIC, J. I. and COLLARD, M. L. (2009). Tql: A query language to support traceability. *TEFSE '09: Proceedings of the 2009 ICSE Workshop on Traceability in Emerging Forms of Software Engineering*. IEEE Computer Society, Washington, DC, USA, 16–20.

MARCUS, A., FENG, L. and MALETIC, J. I. (2003). 3d representations for software visualization. *SoftVis '03: Proceedings of the 2003 ACM symposium on Software visualization*. ACM Press, New York NY USA, 27–ff.

MARCUS, A. and MALETIC, J. I. (2003a). Recovering documentation-to-source-code traceability links using latent semantic indexing. *Proceedings of 25th International Conference on Software Engineering*. IEEE CS Press, Portland Oregon USA, 125–135.

MARCUS, A. and MALETIC, J. I. (2003b). Recovering documentation-to-source-code traceability links using latent semantic indexing. *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*. IEEE Computer Society, Washington, DC, USA, 125–135.

MARCUS, A. and MALETIC, J. I. (2003c). Recovering documentation-to-source-code traceability links using latent semantic indexing. *Proceedings of the International Conference on Software Engineering*. Portland Oregon USA, 125–135.

MASKERI, G., SARKAR, S. and HEAFIELD, K. (2008). Mining business topics in source code using latent dirichlet allocation. *Proceedings of the 1st India software engineering conference*. ACM, New York, NY, USA, 113–120.

MAYRHAUSER, A. V. and VANS, A. (1993). From program comprehension to tool requirements for an industrial environment. *Proceedings of IEEE Workshop on Program Comprehension*. IEEE Comp. Soc. Press, Capri Italy, 78–86.

MCKNIGHT, D. H., C., V. and K., C. (2002). The impact of initial consumer trust on intentions to transact with a web site: a trust building model. *The Journal of Strategic Information Systems*, 11, 297 – 323.

MCMILLAN, C., POSHYVANYK, D. and REVELLE, M. (2009). Combining textual and structural analysis of software artifacts for traceability link recovery. *Traceability in Emerging Forms of Software Engineering, 2009. TEFSE'09. ICSE Workshop on.* IEEE, 41–48.

MULLER, H. A., JAHNKE, J. H., SMITH, D. B., STOREY, M.-A., TILLEY, S. R. and WONG, K. (2000). Reverse engineering: a roadmap. *ICSE '00: Proceedings of the Conference on The Future of Software Engineering.* ACM, New York, NY, USA, 47–60.

OF DAYTON RESEARCH INSTITUTE, I. R. P. U. (1963). *Information Retrieval: Ground Rules for Indexing.* University of Dayton Res. Institute.

OLIVETO, R., GETHERS, M., POSHYVANYK, D. and DE LUCIA, A. (2010). On the equivalence of information retrieval methods for automated traceability link recovery. *Program Comprehension (ICPC), 2010 IEEE 18th International Conference on.* IEEE, 68–71.

PALMER, J. W., BAILEY, J. P. and FARAJ, S. (2000). The role of intermediaries in the development of trust on the www: The use and prominence of trusted third parties and privacy statements. *Journal of Computer-Mediated Communication*, 5.

PEARCE, D. J. and NOBLE, J. (2006). Relationship aspects. *Proceedings of the 5th international conference on Aspect-oriented software development.* New York, NY, USA, 75–86.

PENNINGTON, N. (1987). *Comprehension Strategies in Programming. In: Empirical Studies of Programmers: Second Workshop. G.M. Olsen S. Sheppard S. Soloway eds.* Ablex Publisher Nordwood NJ, Englewood Cliffs, NJ.

PORTER, M. F. (1997). An algorithm for suffix stripping, 313–316.

POSHYVANYK, D., GUÉHÉNEUC, Y.-G., MARCUS, A., ANTONIOL, G. and RAJLICH, V. (2007). Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE Transactions on Software Engineering*, 33, 420–432.

PURDUM, J. (2008). *Beginning C# 3.0: An Introduction to Object Oriented Programming (Wrox Beginning Guides).* Wrox Press Ltd., Birmingham, UK, UK.

RAYNER, K. (1998). Eye movements in reading and information processing: 20 years of research. *Psychological bulletin*, 124, 372.

ROBILLARD, M. P. (2008). Topology analysis of software dependencies. *ACM Trans. Softw. Eng. Methodol.*, 17, 18:1–18:36.

RUMBAUGH, J. (1987). Relations as semantic constructs in an object-oriented language. *SIGPLAN Not.*, 22, 466–481.

SALAH, M., MANCORIDIS, S., ANTONIOL, G. and DI PENTA, M. (2005). Towards employing use-cases and dynamic analysis to comprehend mozilla. *Proceedings of the 21st IEEE International Conference on Software Maintenance*. IEEE Computer Society, 639–642.

SHAO, P. and SMITH, R. K. (2009). Feature location by ir modules and call graph. *Proceedings of the 47th Annual Southeast Regional Conference*. New York, NY, USA, 70:1–70:4.

SHAPIRO, S. and WILK, M. (1965). An analysis of variance test for normality (complete samples). *Biometrika*, 52, 591–611.

SHARIF, B., FALCONE, M. and MALETIC, J. I. (2012). An eye-tracking study on the role of scan time in finding source code defects. *Proceedings of the Symposium on Eye Tracking Research and Applications*. ACM, New York, NY, USA.

SHARIF, B. and KAGDI, H. (2011). On the use of eye tracking in software traceability. *Proceedings of the 6th International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE)*. NY, USA, 67–70.

SHARIF, B. and MALETIC, J. (2010). An eye tracking study on camelcase and under_score identifier styles. *Proceedings of 18th International Conference on Program Comprehension (ICPC)*. IEEE, 196–205.

SHERBA, S. A., A., K. M. and FAISAL, M. (2003). A framework for mapping traceability relationships. *2nd International Workshop on Traceability in Emerging Forms of Software Engineering at 18th IEEE International Conference on Automated Software Engineering*. Montreal, Quebec, Canada, 32–39.

SUN, Y., HE, P. and CHEN, Z. (2004). An improved term weighting scheme for vector space model. *Proceedings of 2004 International Conference on Machine Learning and Cybernetics*. IEEE, vol. 3, 1692–1695.

SUNDARAM, S. K., HAYES, J. H. and DEKHTYAR, A. (2005). Baselines in requirements tracing. *Proceedings of the 2005 workshop on Predictor models in software engineering*. ACM, New York, NY, USA, 1–6.

UWANO, H., NAKAMURA, M., MONDEN, A. and MATSUMOTO, K.-I. (2006). Analyzing individual performance of source code review using reviewers' eye movement. *Proceedings of the 2006 symposium on Eye tracking research & applications (ETRA)*. ACM, New York, NY, USA, 133–140.

WANG, J., PENG, X., XING, Z. and ZHAO, W. (2011). An exploratory study of feature location process: Distinct phases, recurring patterns, and elementary actions. *Proceedings of 27th IEEE International Conference on Software Maintenance (ICSM)*. 213 –222.

WANGA, W., ZENGA, G. and TANG, D. (2010). Using evidence based content trust model for spam detection. *Expert Systems with Applications*, 37, 5599 – 5606.

WEI, X. and CROFT, W. (2006). Lda-based document models for ad-hoc retrieval. *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*. ACM, 178–185.

WILDE, N. and CASEY, C. (1996). Early field experience with software reconnaissance technique for program comprehension. *Proceedings of IEEE Working Conference on Reverse Engineering*.

WOHLIN, C., RUNESON, P., HOST, M., OHLSSON, C., REGNELL, B. and WESSLÉN, A. (2000). Experimentation in software engineering: an introduction.

WU, R., ZHANG, H., KIM, S. and CHEUNG, S. (2011). Relink: recovering links between bugs and changes. *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 15–25.

YUSUF, S., KAGDI, H. and MALETIC, J. (2007). Assessing the comprehension of uml class diagrams via eye tracking. *Proceedings of 15th IEEE International Conference on Program Comprehension (ICPC)*. IEEE, 113–122.

ZHAO, W., ZHANG, L., LIU, Y., SUN, J. and YANG, F. (2006a). Sniafl: Towards a static noninteractive approach to feature location. *ACM Trans. Softw. Eng. Methodol.*, 15, 195–226.

ZHAO, W., ZHANG, L., LIU, Y., SUN, J. and YANG, F. (2006b). Sniafl: Towards a static noninteractive approach to feature location. *ACM Trans. Softw. Eng. Methodol.*, 15, 195–226.

ZOU, X., SETTIMI, R. and CLELAND-HUANG, J. (2010). Improving automated requirements trace retrieval: a study of term-based enhancement methods. *Empirical Software Engineering*, 15, 119–146.

APPENDIX A

DATASETS STATISTICS

In this dissertation, we used eight datasets, which vary in the size, structure, and quality of textual information. To select datasets for our experiments, we define several criteria. First, we select open-source systems, so that other researchers can replicate our experiment. Second, we avoid toy systems that do not represent systems handled by most developers. Yet, both systems were small enough so that we were able to recover and validate their requirements manually in a previous work. Finally, their source code was freely available in their respective SVN repositories. Wherever possible, we utilise datasets developed by other researchers to mitigate threats to validity of our experiments. This appendix describes the summary of all the datasets and how we built oracle for each dataset.

jEdit V4.3[1]**:** It is a text editor for developers written in Java. jEdit includes a syntax highlighter that supports over 130 file formats. It also allows developers to add additional file formats using XML files. It supports UTF-8 and many other encoding techniques. It has extensive code folding and text folding capabilities as well as text wrapping that takes indentation into account. It is highly customisable and can be extended with macros written in BeanShell, Jython, JavaScript, and some other scripting languages. jEdit version 4.3 has 483 classes, measures 109 KLOC, and implements 34 requirements.

JabRef V2.6[2]**:** It is an open-source bibliography reference manager. The native file format used by JabRef is BibTeX, the standard LaTeX bibliography format. JabRef runs on the Java VM (version 1.5 or newer), and should work equally well on Windows, Linux and Mac OS X. Jabref version 2.6 has 579 classes, $287,791$ LOC, and 36 bug reports.

iTrust V10.0[3]**:** It is a medical application that provides patients with a means to keep up with their medical history and records as well as communicate with their doctors, including selecting which doctors to be their primary caregiver, seeing and sharing satisfaction results, and other tasks. iTrust allows the staff to keep track of their patients through messaging capabilities, scheduling of office visits, diagnoses, prescribing medication, ordering and viewing lab results, and so on. iTrust version 10 is developed in Java and it has $19,604$ LOC, 218 files, $3,404$ functions, and 35 requirements.

1. `http://www.jedit.org`
2. `http://jabref.sourceforge.net/`
3. `http://agile.csc.ncsu.edu/iTrust/`

Lucene V3.1 [4]**:** It is an open-source high-performance text-based search-engine library written entirely in Java. It is a technology suitable for nearly any application that requires full-text search, especially cross-platform. Lucene version 3.1 has 434 classes, $111,117$ LOC, and 89 bug reports. We select Lucene $v3.1$ because it contains more closed bugs than other versions and these were linked to its SVN repository.

muCommander V0.8.5 [5]**:** It is a lightweight, cross-platform file manager with a dual-pane interface. It runs on any operating system with Java support. It supports 23 international languages. muCommander supports virtual filesystem as well as local volumes and the FTP, SFTP, SMB, NFS, HTTP, Amazon S3, Hadoop HDFS, and Bonjour protocols. muCommander version 0.8.5 has $1,069$ classes, $124,944$ LOC, and 81 bug reports.

Pooka V2.0 [6]**:** It is an e-mail client written in Java using the JavaMail API. It supports reading e-mails through the IMAP and POP3 protocols. Outgoing emails are sent using SMTP. It supports folder search, filters, and context-sensitive colors. Pooka version 2.0 has 298 classes, weighs 244 KLOC, and implements 90 requirements.

Rhino [7]**:** It is an open-source JavaScript engine entirely developed in Java. Rhino converts JavaScript scripts into objects before interpreting them and can also compile them. It is intended to be used in server-side systems but also can be used as a debugger by making use of the Rhino shell. Rhino version 1.6 has 138 classes, measures 32 KLOC, and implements 268 requirements. Rhino version $1.5R4.1$ has 111 classes, $94,078$ LOC, and 41 bug reports.

SIP V1.0 Communicator [8]**:** It is an audio/video Internet phone and instant messenger that supports some of the most popular instant messaging and telephony protocols, such as AIM/ICQ, Bonjour, IRC, Jabber, MSN, RSS, SIP, Yahoo! Messenger. SIP Communicator has $1,771$ classes, measures 486 KLOC, and implements 82 requirements.

Oracle Building

Ph.D. students manually created traceability links between the requirements of the two systems and their source code classes. They read the requirements and manually looked for classes in the source code that implement these requirements. They used Eclipse to search for the source code. They stored all manually-built links in FacTrace database. Two other Ph.D. persons used FacTrace voting system to accept or reject all manually-built links. At

4. `http://lucene.apache.org`
5. `http://www.mucommander.com/`
6. `http://www.suberic.net/pooka/`
7. `http://www.mozilla.org/rhino/`
8. `http://www.jitsi.org`

no point of the process did we use any automated technique or software repository to create the oracles. This process is the same used in our previous work (Ali *et al.* (2011c,a,b)).

For Lucene, we download bug reports and SVN logs from JIRA repository. Developers usually write bug IDs in SVN logs (Bachmann *et al.* (2010)) when they fix a bug. We automatically extract bug IDs from SVN logs to link bug reports to the classes. For all other datasets, we used the Oracles developed by other researchers (Dit *et al.* (2011b); Eaddy *et al.* (2008a)).

APPENDIX B

LIST OF PUBLICATIONS

The following is a list of our publications related to this dissertation.

Articles in journal or book chapter

– **Nasir Ali**, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. *Trustrace: Mining Software Repositories to Improve the Accuracy of Requirement Traceability Links*, IEEE Transactions on Software Engineering (TSE), to appear.

– **Nasir Ali**, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. *Factors Impacting the Inputs of Traceability Recovery Approaches*, chapter 7. Springer, September 2011. Note: 28 pages.

Conference articles

– **Nasir Ali**, Zohreh Sharafi, Yann-Gaël Guéhéneuc, and Giuliano Antoniol, *An Empirical Study on Requirements Traceability Using Eye-Tracking.* In Massimiliano Di Penta and Jonathan I. Maletic, editors, Proceedings of the 28th International Conference on Software Maintenance (ICSM), September 2012. IEEE Computer Society Press.

– **Nasir Ali**, Aminata Sabané, Yann-Gaël Guéhéneuc, and Giuliano Antoniol, *Improving Bug Location Using Binary Class Relationships.* In Mariano Ceccato and Zheng Li, editors, Proceedings of the 12th International Working Conference on Source Code Analysis and Manipulation (SCAM), September 2012. IEEE Computer Society Press.

– **Nasir Ali**, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. *Requirements Traceability for OO systems by Partitioning Source Code.* In Proceedings of the 18th Working

Conference on Reverse Engineering (WCRE'11), October 17-20, 2011. IEEE Computer Society Press.

– **Nasir Ali**, Wei Wu, Giuliano Antoniol, Massimiliano Di Penta, Yann-Gaël Guéhéneuc, and Jane H. Hayes. *MoMS: Multi-objective Miniaturization of Software.* In James R. Cordy and Paolo Tonella, editors, Proceedings of the 27th International Conference on Software Maintenance (ICSM), September 2011. IEEE Computer Society Press.

– **Nasir Ali**, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. *Trust-based Requirements Traceability.* In Proceedings of the 19th International Conference on Program Comprehension (ICPC), 22 - 24 June, 2011. IEEE Computer Society Press.

– **Nasir Ali**. Trustrace: *Improving Automated Trace Retrieval Through Resource Trust Analysis.* In Proceedings of the 19th International Conference on Program Comprehension (ICPC), 22 - 24 June, 2011. IEEE Computer Society Press.

Following is a list of our publications, in which I participated during my Ph.D., related to software maintenance and comprehension.

Articles in journal

– Abdou Maiga, **Nasir Ali**, Marwen Abbes, Foutse Khomh, Yann-Gaël Guéhéneuc, Nesa Asoudeh, Yvan Labiche, (2012) *An Empirical Study of the Impact of Blob and Spaghetti Code Antipatterns On Program Comprehension*, Journal of Empirical Software Engineering (EMSE) (submitted).

Conference articles

– Abdou Maiga, **Nasir Ali**, Neelesh Bhattacharya, Aminata Sabane, Yann-Gaël Guéhéneuc, Esma Aïmeur, (2012) *SMURF: A SVM-based, Incremental Antipattern Detection Ap-*

proach, In Proceedings of the 19th Working Conference on Reverse Engineering (WCRE), October 15-18, Kingston, Ontario (Canada). IEEE Computer Society Press.

– Abdou Maiga, **Nasir Ali**, Neelesh Bhattacharya, Aminata Sabane, Yann-Gaël Guéhéneuc, Giuliano Antoniol, Esma A"imeur, (2012) *Support Vector Machines for Antipattern Detection*, Proceedings of the 27th IEEE International Conference on Automated Software Engineering (ASE'12), 3-7 September 2012, IEEE Computer Society Press.