Using and Characterizing Change-sets to Support Feature Location

Muslim Chochlov

Lero

Department of Computer Science and Information Systems
The Faculty of Science and Engineering
University of Limerick

Submitted to the University of Limerick for the

Degree of Doctor of Philosophy (PhD) year 2017
1. Supervisor: Dr. Jim Buckley  
Department of Computer Science and Information Systems  
University of Limerick  
Ireland

2. Supervisor: Dr. Michael English  
Department of Computer Science and Information Systems  
University of Limerick  
Ireland

1. Reviewer: Dr. Patrick Healy  
Department of Computer Science and Information Systems  
University of Limerick  
Ireland

2. Reviewer: Dr. Jens Knodel  
The Fraunhofer Institute  
Germany

1. Chair: Dr. Chris Exton  
Department of Computer Science and Information Systems  
University of Limerick  
Ireland

Day of the defence: 31 October 2017
Abstract

Feature location is finding the source code that implements specific functionality in software systems. Feature location is a complex activity and, when performed manually, it may require significant developers’ effort. Consequently, semi-/automated feature location techniques have been proposed to assist developers. One common group of such approaches utilizes textual information in source code, and applying information retrieval techniques. Since there is a paucity of meaningful terms in source code, a reasonable research direction is to mix various data sources to expand upon the dataset of meaningful terms in source code entities, for information retrieval. One such data source is the set of change-set descriptions. Not much work has been done in the area of meaningful term expansion using change-set descriptions and the extent to which such expansions are useful has not been thoroughly studied in the literature.

This work proposes a technique which leverages change-set data sets as a source of meaningful terms that can act as source code descriptors (ACIR). It is the first work to study change-sets in such a role in isolation and characterize their effectiveness as a data-set for information retrieval based feature location. Specifically, it characterizes the performance of ACIR in terms of granularity, recentness of change-sets, aggregation of recent change-sets by change request, and filtering of “management” change-sets using textual classification via a custom built tool, implementing ACIR. The evaluation work is larger than the other works in this area, employing 8 different subject systems with a total of 600 re-enactment samples.
It was found, for ACIR, that the effort required to locate entities is, in general, lower at method level than file level of granularity. Additionally, using more recent change-sets improves the effectiveness of ACIR. However, aggregation of recent change-sets by change request, decreases effectiveness. Surprisingly, it was also found, that text classification based filtering of “management” change-sets, based on generic management terms, decreases the effectiveness of ACIR.

Further, the findings indicate that certain characteristics of subject systems seem to affect the performance of ACIR: a strongly pronounced dichotomy of subject systems emerged, where one set recorded better feature location using ACIR and another recorded better FL using a more traditional baseline approach. Finally, it was found, that merging ACIR and a baseline approach significantly improves performance over the baseline approach by 95% and over ACIR alone by 17%.

Apart from the more concrete findings on the effectiveness of the newly proposed technique itself, the most fundamental finding is the importance of rigorously characterizing proposed feature location techniques, to identify their optimal configurations. The results also suggest it is important to characterize the software systems under study when selecting the appropriate feature location technique. In the past, configuration of the techniques and characterization of subject systems have not been considered first-class entities in research papers, whereas the results presented here suggests these factors can have a big impact.
Declaration

I herewith declare that I have produced this work without the prohibited assistance of third parties and without making use of aids other than those specified; notions taken over directly or indirectly from other sources have been identified as such. This work has not previously been presented in identical or similar form to any other Irish or foreign examination board.

The thesis work was conducted from year 2013 to year 2017 under the supervision of Dr. Jim Buckley and Dr. Michael English at University of Limerick.

Limerick, year 2017
Acknowledgements

First, I would like to thank my two great supervisors Dr. Jim Buckley and Dr. Michael English. Their advice, patience, and wisdom helped me to achieve the goals of this research. Without their support, this path would be a lot more difficult to walk.

I am especially grateful to my wife, Ilona. She was extremely supportive and encouraged me on many occasions. She also proof read the entire work and was my first objective and adorable critic.

The final thanks and dedication of this work goes to our lovely son, Mark. He came to this world during the final stages (and the most difficult) of writing the dissertation and his “cooing” was the most supportive sound to me.

This work was supported, in part, by Science Foundation Ireland Grants 12/IP/1351 and 13/RC/2094 to Lero the Irish Software Engineering Research Centre (http://www.lero.ie).
Publications


### Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACIR</td>
<td>Aggregation of Change-set descriptions for Information Retrieval</td>
</tr>
<tr>
<td>AP</td>
<td>Average precision</td>
</tr>
<tr>
<td>API</td>
<td>Application programming interface</td>
</tr>
<tr>
<td>BL</td>
<td>Bug localization/location</td>
</tr>
<tr>
<td>FL</td>
<td>Feature location</td>
</tr>
<tr>
<td>FLT</td>
<td>Feature location technique</td>
</tr>
<tr>
<td>IA</td>
<td>Impact analysis</td>
</tr>
<tr>
<td>IR</td>
<td>Information retrieval</td>
</tr>
<tr>
<td>ITS</td>
<td>Issue tracking system</td>
</tr>
<tr>
<td>LDA</td>
<td>Latent Dirichlet allocation</td>
</tr>
<tr>
<td>LSI</td>
<td>Latent semantic indexing</td>
</tr>
<tr>
<td>MAP</td>
<td>Mean average precision</td>
</tr>
<tr>
<td>MRR</td>
<td>Mean reciprocal rank</td>
</tr>
<tr>
<td>NLP</td>
<td>Natural language processing</td>
</tr>
<tr>
<td>PM</td>
<td>Pattern matching</td>
</tr>
<tr>
<td>SOA</td>
<td>Service oriented architecture</td>
</tr>
<tr>
<td>Acronym</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td><strong>SVN</strong></td>
<td>Subversion version control system</td>
</tr>
<tr>
<td><strong>TF-IDF</strong></td>
<td>Term frequency - inverse document frequency</td>
</tr>
<tr>
<td><strong>TFLT</strong></td>
<td>Textual feature location technique</td>
</tr>
<tr>
<td><strong>TFLT(_B)</strong></td>
<td>The baseline information retrieval based textual feature location technique.</td>
</tr>
<tr>
<td><strong>TFLT(_{IR})</strong></td>
<td>Information retrieval based textual feature location techniques</td>
</tr>
<tr>
<td><strong>TLR</strong></td>
<td>Traceability link recovery</td>
</tr>
<tr>
<td><strong>VCS</strong></td>
<td>Version control system</td>
</tr>
<tr>
<td><strong>VSM</strong></td>
<td>Vector space model</td>
</tr>
</tbody>
</table>
## Contents

Abstract iii  
Declaration v  
Acknowledgements vi  
Publications vii  
Abbreviations viii  
Contents x  
List of Tables xvi  
List of Figures xviii

1 Introduction  
1.1 Features and Feature Location 1  
1.2 The Motivation for the Automation of Feature Location 4  
1.3 Feature Location Techniques and Their Types 5  
1.4 IR-based Textual Feature Location Techniques 7  
1.5 The Research Gap 9  
1.6 Objectives 10  
1.7 Research questions 11  
1.8 Contributions 12  
1.9 Structure 13
# 2 Literature Review

## 2.1 Literature review process ........................................ 15

### 2.1.1 Phase 1: Identifying the State-of-the-art FLTs .......... 16

#### 2.1.1.1 Existing Literature Reviews of Feature Location Techniques ............................... 16

#### 2.1.1.2 The Literature Search Protocol Used to Update The Existing FL Reviews ....................... 20

#### 2.1.1.2.1 Using Digital Libraries to Find FLT Papers 20

#### 2.1.1.2.2 The Inclusion and Exclusion Criteria for FLT Papers 24

### 2.1.2 Phase 2: Identifying Approaches that Leverage Change-sets 25

#### 2.1.2.1 The Areas Related to FL: BL, IA, and TLR .... 25

#### 2.1.2.2 The Literature Search Protocol for Relevant BL, IA, and TLR Techniques 26

## 2.2 Overview of FLTps .................................................. 27

### 2.2.1 The Existing Taxonomies of FLTps .......................... 28

### 2.2.2 The Taxonomy of FLTps Used in This Work ............... 28

#### 2.2.2.1 Populating the Taxonomy of FLTps .................. 30

#### 2.2.2.2 Dynamic FLTps ........................................ 31

#### 2.2.2.3 Static Structural FLTps ............................... 34

#### 2.2.2.4 Hybridization of Dynamic and Structural FLTps ..... 36

### 2.2.3 Static Textual FLTps ........................................... 36

#### 2.2.3.1 Pattern Matching in TFLTps .......................... 38

#### 2.2.3.2 Natural Language Processing in TFLTps .......... 39

#### 2.2.3.3 Information Retrieval in TFLTps .................... 40

#### 2.2.3.4 Common Steps in TFLTIRs ............................. 41

#### 2.2.3.5 Hybridization of TFLTps with Other Static and Dynamic FLTps ............................ 46

#### 2.2.3.6 The Advantages and Disadvantages of TFLTps .. 49

### 2.2.4 Trends in Types of Techniques Going Forward ........... 50

### 2.3 How FLTps are Evaluated and The Associated Problems .... 50

#### 2.3.1 Identification of the Gold Set ............................. 51

#### 2.3.2 The Problems of Evaluating FLTps ....................... 52
CONTENTS

2.3.3 Metrics Used in the Evaluation of FLTs

2.4 TFLT_{IRs}: Problems in the Area

2.4.1 Identifying Best-Performing Configurations in FLTs

2.4.2 The Vocabulary Paucity Problem in TFLT_{IRs}

2.5 How the Change-set Descriptions were Used in FL, BL, IA, and TLR

2.5.1 Change-sets as a Data Source

3 Aggregating Change-sets for Feature Location

3.1 Overview

3.2 The ACIR Approach

3.3 Details of Steps in the ACIR Technique

3.3.1 The Running Example

3.3.2 Partitioning of the Source Code

3.3.3 Annotating Source Code Entities with Change-set Descriptions

3.3.3.1 ACIR Configurations: Grouping and Classification of Change-Sets

3.3.3.2 Obtaining the Change-sets Configurations

3.3.3.3 Aggregating Change-sets by Granularity and Most-recent/All Historical Change-sets

3.3.3.4 Aggregating Change-sets of a Change Request

3.3.3.5 Aggregating Change-sets by Type

3.3.3.6 Representing Source Code Entities with Change-set Descriptions

3.3.3.7 Example: Annotating the Method in the Running Example Using Various Configurations
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>The amended list of venues used in the current literature search</td>
<td>23</td>
</tr>
<tr>
<td>2.2</td>
<td>The statistics of FL papers according to the classification used in</td>
<td>31</td>
</tr>
<tr>
<td></td>
<td>this work.</td>
<td></td>
</tr>
<tr>
<td>2.3</td>
<td>The VSM matrix of the five example documents.</td>
<td>32</td>
</tr>
<tr>
<td>2.4</td>
<td>Change requests that could be traced from change-sets.</td>
<td>67</td>
</tr>
<tr>
<td>2.5</td>
<td>FLT's using change-sets</td>
<td>72</td>
</tr>
<tr>
<td>2.6</td>
<td>Related work</td>
<td>78</td>
</tr>
<tr>
<td>3.1</td>
<td>The set of change-set descriptions in the running example and their</td>
<td>85</td>
</tr>
<tr>
<td></td>
<td>bag-of-words representations.</td>
<td></td>
</tr>
<tr>
<td>3.2</td>
<td>Values of classification/grouping of change-sets</td>
<td>87</td>
</tr>
<tr>
<td>3.3</td>
<td>Change-sets' aggregation support in VCSs: Git, SVN, and CVS</td>
<td>91</td>
</tr>
<tr>
<td>3.4</td>
<td>Example: Annotating the Method in the Running Example Using Various</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>Configurations</td>
<td></td>
</tr>
<tr>
<td>4.1</td>
<td>Subject systems of the pilot study</td>
<td>107</td>
</tr>
<tr>
<td>4.2</td>
<td>Subject Systems</td>
<td>112</td>
</tr>
<tr>
<td>5.1</td>
<td>ACIR Descriptive Statistics</td>
<td>123</td>
</tr>
<tr>
<td>5.2</td>
<td>Comparing ACIR with Previous Work</td>
<td>126</td>
</tr>
<tr>
<td>5.3</td>
<td>Effort for File vs Method Level Cases of Granularity</td>
<td>127</td>
</tr>
<tr>
<td>5.4</td>
<td>Effectiveness for All Historical vs Recent Cases of change-set Range</td>
<td>129</td>
</tr>
<tr>
<td>5.5</td>
<td>MAP for ACIR configurations and TFLT&lt;sub&gt;B&lt;/sub&gt;</td>
<td>135</td>
</tr>
<tr>
<td>5.6</td>
<td>The effort required when finding a file and method expressed in</td>
<td>136</td>
</tr>
<tr>
<td></td>
<td>MRR(%) along with Wilcoxon significance test</td>
<td></td>
</tr>
<tr>
<td>Table</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>-------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>5.7</td>
<td>Wilcoxon signed rank statistical test of MAP gains for ACIR&lt;sub&gt;A&lt;/sub&gt; and ACIR&lt;sub&gt;R&lt;/sub&gt;</td>
<td></td>
</tr>
<tr>
<td>5.8</td>
<td>Wilcoxon signed rank statistical test of the effects of aggregation of recent change-sets by change request</td>
<td></td>
</tr>
<tr>
<td>5.9</td>
<td>Wilcoxon signed rank statistical test of filtering effects</td>
<td></td>
</tr>
<tr>
<td>5.10</td>
<td>Wilcoxon signed rank statistical test of MAP gains for eight subject systems</td>
<td></td>
</tr>
<tr>
<td>5.11</td>
<td>The average times of initial search corpus creation.</td>
<td></td>
</tr>
<tr>
<td>6.1</td>
<td>Similar features of subject systems</td>
<td></td>
</tr>
<tr>
<td>6.2</td>
<td>MAP for ACIR&lt;sub&gt;R&lt;/sub&gt;, TFLT&lt;sub&gt;B&lt;/sub&gt;, and BACIR&lt;sub&gt;R&lt;/sub&gt;</td>
<td></td>
</tr>
<tr>
<td>6.3</td>
<td>Wilcoxon signed rank statistical test of TFLT&lt;sub&gt;B&lt;/sub&gt;, ACIR&lt;sub&gt;R&lt;/sub&gt;, and BACIR&lt;sub&gt;R&lt;/sub&gt; MAP distributions</td>
<td></td>
</tr>
<tr>
<td>6.4</td>
<td>Pairwise comparisons using Wilcoxon signed rank test</td>
<td></td>
</tr>
<tr>
<td>6.5</td>
<td>The average times of initial search corpus creation (method level).</td>
<td></td>
</tr>
</tbody>
</table>
List of Figures

1.1 The relationship between the features and their implementation in source code. .............................................. 3
1.2 The focus of this research. ........................................ 10
2.1 The literature review process ....................................... 17
2.2 The comparison of the taxonomies of FLTs ..................... 30
2.3 Venn diagram showing relations between gold set and result set 51
2.4 Example: a structure of Git VCS. ................................. 63
2.5 An example commit of JGit. ........................................ 64
2.6 The common types of change-sets as used in different works and the commonality between these types. .................. 65
2.7 An example commit of MariaDB. ................................. 65
2.8 Change-sets and source code ties. ............................... 69
2.9 Multi-language systems ........................................... 70
3.1 The ACIR approach .................................................. 82
3.2 Example: code changes in print method ........................ 84
3.3 Example: applying diff to compare the change-set (02008c3d with the change-set (3ac3d3a6). ....................................... 92
3.4 History of changes for lines of code in print method, from the running example. .................................................. 93
3.5 Generic change-set aggregation approach ....................... 94
3.6 Aggregating by branches .......................................... 96
3.7 Aggregating by author and period between change-sets ....... 97
3.8 The baseline TFLT_B approach .................................. 101
4.1 Evaluation procedure .......................................................... 109
4.2 Change request work-flow and associated change-sets ............... 113
4.3 Division of change-sets into annotation and test sets ................. 114

5.1 The effort of different levels of granularity: a) when the method level effectiveness is adjusted; b) when the file level effectiveness is adjusted................................................................. 128
5.2 The effectiveness of different change-set ranges: a) in Rhino; b) in Mylyn.Tasks................................................................. 130
5.3 Distribution of average precision data: x-scale - average precision, y-scale - density ................................................................. 134

6.1 BACIR\_R approach ............................................................ 155
6.2 Distribution of average precision data: x-scale - average precision, y-scale - density ................................................................. 158
1

Introduction

1.1 Features and Feature Location

In software systems, the term “feature” usually describes a distinctive characteristic of those systems that is implemented in source code [IEEE Standard 829-1998]. In this work however, a feature refers to a functionality that is defined by the requirements of the software system and is accessible to developers and users [Dit et al. 2013a].

While features describe the functionality a software system provides, the software architecture (and its substructures such as layers and components) defines the macro-structures across which these features are implemented. Consequently, a feature can be de-localized in an architecture: that is, it can have its elements located in different layers/components. Though there is a certain orthogonality between features and software architecture, the latter can provide clues as to where features are implemented. For example, imagine a system with a user-interface layer, a business-logic layer and a database layer. It is likely that most features will have elements in all three layers.

Features can be characterized as:

- Abstract (a feature is not a physical source code entity, but a higher level, more distributed concept).

- Functional in that end-users see them as real world facilities, that are implemented in software systems.
1.1 Features and Feature Location

- Observable: removing or adding a feature - and hence its implementing code - will result in changes to a software product’s state or behaviour.

Features are implemented in source code, using one or more source code entities, where a source code entity is a specific piece of code such as statement, variable, sub-program, class, file, package, etc. Source code entities are said to have a “level of granularity” (e.g. line level granularity (fine granularity), file level granularity (coarse granularity)). In between these two levels of granularity is the sub-program level of granularity - in this work, a sub-program refers to any callable source code entity such as procedure, subroutine, method, or function. However, when put into a specific programming context a more appropriate term is used: for example, “method” is used when referring to Java[1] sub-programs.

It may be that the mapping between features and the set of source code entities happens at many levels for each feature. For example, a feature may be mapped to several methods, a control block and a separate class. The entire set of source code entities that implement the feature is referred to as the extent of a feature.

Figure 1.1 illustrates the relationship between the features and their implementation in source code, with features on the left and source code entities on the right.

There is a many-to-many relationship between features and source code entities. The edges, that connect the feature-set and the source code entities set in the figure, express the “belongs to” relationships: a source code entity belongs to a feature(s). Finding the entities that belong to a feature is the task of feature location (FL). The following definition for feature location is used in this work:

**Definition:** Feature location is mapping a feature to source code entity(ies) that implement that feature.

There are several activities, related to FL: concept location, bug localization/location (BL), impact analysis (IA), and traceability link recovery (TLR). Concept and feature location are often used interchangeably in the literature [Dit et al.](https://java.com/en/download/)
But concepts are considered as stretching beyond observable system functionality to any concern that can be cohesively expressed about a software system. For example, if a developer was charged with moving a system to a new platform, they might be concerned with the platform interface. In this instance their concern location task would be to identify where the system “talked” to the platform and not concerned with the system’s functionality. Thus, feature location is a subset of concept location. Bug localization locates faulty source code entities as part of corrective software maintenance activity (Lukins et al., 2008). IA evaluates the change propagation and change “ripple effect”, resultant from code edits (Bohner and Arnold, 1996). TLR aims to discover, sustain, and repair the links between the documentation and the source code entities (Antoniol et al., 2002).
1.2 The Motivation for the Automation of Feature Location

The need for feature location usually arises when a developer is unfamiliar with all or part of the source code (Bourque and Fairley, 2014; Lientz and Swanson, 1980). One common scenario is when a new developer joins a software project and needs to locate features as part of their software maintenance activities. For example, a developer may be given a change request that involves modifications to a certain feature. In this case, they must first locate the feature, before they can make the changes required.

Indeed, given the scale of today’s software systems, even a more experienced developer would not be expected to know the location of all the features that the system implements, and so they too would be mandated with feature location. Feature location for such developers becomes more difficult as the scale and complexity of the system increases. It becomes harder again if documentation and/or knowledgeable colleagues are unavailable (Roehm et al., 2012; Wang et al., 2011).

Another common scenario for organizations is the modernization of legacy systems (Bennett and Rajlich, 2000) and feature location can be important in this also (Rubin and Chechik, 2013). For example, the organization might want to reduce the amount of code they need to port to a new technology by removing certain unused functionality in the system: this requires the identification of the code associated with that unused functionality. Alternatively they may wish to move to a service-oriented system where individual services interact to form the whole. If a service is considered an end-user functionality, then the first part of this re-engineering task is to identify the implementation of those functionalities in the existing code-base.

Manual feature location can be difficult, because it requires a developer to inspect often numerous source code entities and the relationships between these entities, and to understand the workings behind these source code entities with respect to the application domain (an activity often associated with program comprehension (Ko et al., 2006; Pennington, 1987; Singer et al., 2010; Wang et al., 2011)). The research suggests (Pennington, 1987) that the short term memory of a developer’s brain is used to build and refine an abstract model of a software
system, that describes the source code entities and their relationships in terms of how that system functions in the application domain (Hatton, 1998; Mayrhofer and Vans, 1995; O’Brien et al., 2004; Rajlich and Wilde, 2002; Shneiderman and Mayer, 1979; Siegmund et al., 2014). Given the number of entities involved in such a task (various source code entities at various levels of granularity, application domain concepts), natural physiological limitations of the human brain seem to apply when a developer is locating features (Miller’s magical number seven (Miller, 1956)).

Given the alignment between program comprehension and feature location (Ko et al., 2006; Marcus and Haiduc, 2013; Singer et al., 2010; Wang et al., 2011), it seems that feature location is frequent in software maintenance (Dit et al., 2013a). Software maintenance activities are expensive: independent studies suggest that over 60% of the software development budget are software maintenance costs (Brooks, 1975, 1995; Canfora and Cimitile, 2001). Program comprehension is a major part of these software maintenance activities, reported by various case studies as consuming over 50% of developers’ maintenance effort (Canfora and Cimitile, 2001; Fjeldstad and Hamlen, 1983). Hence, it can be concluded that feature location is an effort intensive, difficult task.

To address the difficulty of manual feature location and the costs that arise because of that difficulty, researchers have concentrated their efforts on automating feature location.

1.3 Feature Location Techniques and Their Types

In the literature, feature location techniques (FLTs) are often associated with finding the initial entry point to a feature (Dit et al., 2013a), but some authors also use FLTs to identify the extent of a feature (Jordan et al., 2015; Kastner et al., 2014). In this work, a more relaxed definition for a FLT is used, that stems from the definition of a feature location in Section 1.1.

Definition: A feature location technique is a semi-automated approach to feature location, where the goal is to identify at least one relevant source code entity, but could also be to identify the entire extent of a feature.
1.3 Feature Location Techniques and Their Types

Because deriving a full “feature location algorithm” does not seem possible (Biggerstaff [1994]), approximation approaches are employed for FLTs. A strategy, often employed by developers, is to search for clues in source code until enough evidence is obtained to map a feature to source code entity(ies) (Soloway and Ehrlich [1984]). FLTs attempt to imitate this behaviour by applying various types of analysis to software systems towards this evidence: usually they inspect the software system’s execution traces (when the feature is exercised), the source code for clues, or/and any auxiliary data to provide evidence towards FL.

The type of analysis is so central to FLTs that it was used to classify the FLTs in two well-known papers, describing FLTs en-masse (Dit et al. [2013a]; Rubin and Chechik [2013]). Dit et al. (2013a) suggest that there are three major types of analysis utilized by FLTs: dynamic, static, and textual; whereas Rubin and Chechik (2013) suggest that there are two such types: dynamic and static. Likewise, in this work, the type of analysis is used to denote the type of a FLT. For instance, if a FLT utilizes dynamic analysis it is referred to as a dynamic FLT. Because the classification of FLTs differs by authors, the classification used is explicitly mentioned in the text. Below, the three major types of FLTs are described, according to the classification by Dit et al. (2013a):

- Dynamic FLTs examine execution traces collected at run-time and thus record source code entities that might be relevant to a feature (Biggerstaff, 1994; Eisenbarth et al., 2003; Savage et al., 2010; Wilde and Scully, 1995). To eliminate a large portion of irrelevant source code entities that would naturally arise, a scenario or a test case, that does exercise the feature, is prepared by a domain expert and another, often highly similar test case, that doesn’t exercise the desired functionality is also created. This execution trace of the latter can then be subtracted from the first execution trace resulting in a smaller subset of a source code executed when exercising the feature.

- Static FLTs explore the control/data flow graph of a system. Such a technique would usually start at a node in the graph (representing a source code entity) and would traverse the graph to find all reachable, relevant nodes and map them to a feature (Kazato et al., 2013a; Petrenko and Rajlich, 2013a).
1.4 IR-based Textual Feature Location Techniques

Picking an arbitrary starting node could lead to the exploration of a large number of irrelevant nodes. Hence, a common improvement is to meaningfully determine a starting node beforehand to decrease the search space.

- **Textual FLT**s (TFLTs) leverage potentially meaningful textual data in source code (for example, variable/sub-program/class names) and usually utilize pattern matching (PM), information retrieval (IR), and natural language processing (NLP) (Dit et al., 2013a). IR-based TFLTs (TFLT\textsubscript{IR}) are particularly often studied since they offer a reasonable trade-off between the complexity of NLP approaches and the naive simplicity of PM (Marcus et al., 2004), and it is this approach that is focused on in this thesis.

Finally, a combination of different types of analysis could be used in FLT\textsubscript{s} (Eisenbarth et al., 2003; Kazato et al., 2013b; Petrenko and Rajlich, 2013; Poshyvanyk et al., 2012; Scanniello and Marcus, 2011). In this work, such FLT\textsubscript{s} are called hybrid FLT\textsubscript{s}.

**1.4 IR-based Textual Feature Location Techniques**

In TFLT\textsubscript{IR}, when a user submits a search query, the most relevant source code entities are retrieved and ranked according to their similarity to the query. In fact, meaningful words that appear in source code are used to construct textual documents (Dit et al., 2013a; Marcus et al., 2004) and these textual documents are compared to the query for similarity. In a nutshell, TFLT\textsubscript{IR}s parse source code files, retrieve meaningful words for each source code entity, use those words to construct textual documents representing these source code entities, store the textual documents, and usually employ indexing mechanisms for fast retrieval and comparison of the textual documents with the query.

IR techniques were initially designed to search collections of large text documents (Salton and McGill, 1983). Therefore, such approaches rely on the abundance of meaningful words. In contrast, source code entities are usually represented by short textual documents, as a large proportion of the source code entities are made up of keywords and only specific elements of source code allow for
meaningful lexicons (for example, sub-program names, class names, comments). The *vocabulary paucity problem* (the lack of meaningful terms in source code) and, additionally, the vocabulary mismatch problem (that query terms do not always match the terms found in source code) makes it hard for traditional IR approaches to retrieve relevant source code entities, especially if that retrieval is focused on finer levels of granularity, like sub-programs (where the vocabulary paucity problem becomes more exacerbated). Existing TFLTts try to address these problems in the following ways:

- Expanding the search query by suggesting meaningful keywords (Hill et al., 2009, 2014b).
- Assisting a developer to navigate the lengthy result set (Petrenko and Rajlich, 2013; Scanniello and Marcus, 2011).
- Propagating correct source code entities towards the beginning of the ranked result sets (Bassett and Kraft, 2013; Hill et al., 2011).

Surprisingly, not much work has been done on the expansion of the vocabularies associated with source code entities. One alternative to provide potentially richer source of textual information comes from data available in *version control systems* (VCS), *issue tracking systems* (ITS), and various developer-communication tools. Given the richer textual descriptions that these tools employ, it may be advantageous to use these systems for TFLTIR (Chen et al., 2001; Cleary and Exton, 2007; Cubranic et al., 2005; Kevic and Fritz, 2014; Ratantayon et al., 2010b; Würsch et al., 2013; Zamani et al., 2014). Particularly, the change-sets of VCSs seem to be interesting, because their textual descriptions explain the rationale for changes, often in terms of the application domain, and because they are explicitly connected to source code (Chen et al., 2001; Ratantayon et al., 2010b). In this work:

**Definition:** A change-set is an atomic change, recorded in VCS, that touched the file(s) of a software system.
1.5 The Research Gap

And features are located in source code files only. Hence, when using change-sets to locate features, non-source code files, that are part of a change-set, are ignored. But as an aside, it should also be noted that, in contemporary multi-language software systems change-set descriptions, in general, could be more holistically applied across documentation, descriptor files and other system artefacts, because of their (English) language source and granularity independence.

1.5 The Research Gap

The analysis of two existing papers, describing FLTs en-masse (Dit et al., 2013a; Rubin and Chechik, 2013), and a subsequent literature review revealed that change-sets have been used to some extent in FLTs. In specific, among the FLT papers identified by Dit et al. (2013a)\(^1\), there were eight papers that leverage non-source code textual data. In an additional literature review carried out as part of this PhD, another 27 FLT papers published from January 2011 and going to April 2015 were identified (see Section 2.1). Of those 27, three papers described FLTs that leverage non-source code textual data. In these 11 FLT papers, four unique approaches were found that leverage change-set descriptions. The literature search was then expanded to include the relevant papers of BL, IA, and TLR. Another two approaches were found that leverage change-set descriptions for IA.

Yet the application of change-sets in these six techniques (4 FLTs and two IA approaches) was limited and the approach that leverages change-set descriptions to address the vocabulary paucity problem for TFLT\(_{IR}\) has never been adequately described in the literature. Particularly, the existing approaches seem to use inappropriate granularity (Canfora and Cerulo, 2006; Chen et al., 2001), utilize change-set descriptions as a small part of their process (Canfora and Cerulo, 2006; Zanjani et al., 2014), focus on a smaller sub-set of source code entities (Canfora and Cerulo, 2006; Zanjani et al., 2014), or use several data sources that hide the value of each individual source contribution (Zanjani et al., 2014). In contrast,\(^1\)

\(^1\)This literature review is used to cover the FLTs up to February 2011. The authors of the other review, Rubin and Chechik (2013), claim that 22 of 24 FL approaches reviewed in their survey are covered by Dit et al. (2013a).
1.6 Objectives

Figure 1.2: The focus of this research.

the approach proposed in this work is based on a granularity, determined as appropriate by an empirical study, in terms of a real-world scenario. It also uses the entire set of source code entities for FL and uses change-set descriptions as a single data source in isolation.

In addition, the potential role of change-set descriptions as an alternative textual data source in this area has not been thoroughly studied in isolation, but has only been studied when incorporated as part of a hybrid technique. Consequently, existing work does not provide a thorough experimental assessment of the usefulness of this data source. Such a study would allow researchers to assess the best usage of these sources and whether they are better-than or complimentary-to the data from source code. For example, if the results are good, but different to those generated when analysing source code lexicons, it is an indicator that it might hybridize well with the source code technique.

To summarize, the focus of this work is to investigate if and how the change-set descriptions could be leveraged to address the vocabulary paucity problem in TFLT\textsubscript{IR}, as shown in Figure 1.2.

1.6 Objectives

Following from the research gap, the main objective of this work is to leverage change-sets for TFLT\textsubscript{IR} and to characterize the effectiveness of such an approach.
To accomplish this objective the following sub-objectives were set:

- To derive an approach that leverages change-set descriptions to address the vocabulary paucity problem in TFLT\textsubscript{IR}. The resultant approach is called ACIR (Aggregation of change-set descriptions for Information Retrieval).

- To empirically assess and optimize ACIR (identify the best performing configuration of the approach).

- To compare the optimized ACIR against a state-of-the-art baseline TFLT\textsubscript{IR} (TFLT\textsubscript{B}).

The comparison of the best-performing configuration of ACIR to the TFLT\textsubscript{B} identified a dichotomy: half of the subject systems in the experiment responded better to ACIR and the other half responded better to TFLT\textsubscript{B}. Two post-hoc objectives were formulated based on this finding:

- To characterize the two sets of subject systems, to determine any differences that might be related to their response to ACIR and TFLT\textsubscript{B}.

- To evaluate the potential for hybridization of the best-performing configuration of ACIR and TFLT\textsubscript{B}.

1.7 Research questions

The following research question encompasses the direction of this research:

| Can the annotation of code by change-set descriptions assist FL? |

The more detailed sub-questions answered in this work include:

**RQ1:** How can change-sets be leveraged to improve TFLT\textsubscript{IR}s?

**RQ2:** What is the best configuration towards such improvement of TFLT\textsubscript{IR}s in terms of change-sets recentness, type, and granularity?
1.8 Contributions

**RQ3:** How does best-practice configuration of the new approach (ACIR), as defined by the answer to the second question, compare to a state-of-the-art baseline TFLT\textsubscript{B} approach that leverages meaningful lexicons in source code?

The results from research question 3, suggested that some systems react well to ACIR and others to TFLT\textsubscript{B}. This suggests a characterization of systems in terms of the FLT which might be appropriately applied to them and that a hybrid approach might be beneficial. Hence another two research questions assess:

**RQ4:** What system characteristics respond well to ACIR as opposed to TFLT\textsubscript{B}?

**RQ5:** How efficient is the hybridization of source code and change-set textual sources in an IR approach?

1.8 Contributions

The original contributions of this work to the existing knowledge include:

- A novel change-set-based, IR, FL technique - ACIR.

- Empirically determining the best configuration of ACIR with respect to granularity of source code entities, age of change-sets, and type of change-sets.

- Partially characterizing its applicability in terms of system characteristics.

- The implementation of ACIR, its configurations, the hybrid approach, and TFLT\textsubscript{B} as open source tooling, available to other researchers.

The specific findings of this work suggest that certain configurations of ACIR result in a better performance. A broader implication (and contribution) for FLT research is the need to characterize FLTs in general, because, as was the case in this work, the optimal configuration may significantly outperform other configurations. Currently, and in contrast, the research direction of FLTs has geared towards combinations of approaches and new approaches like query expansion. This
work suggests that when implementing such approaches, practitioners should also concentrate on determining the best performing configuration and subsequently ignore less optimal configurations.

Likewise, it was also found that characteristics of software systems have a significant impact on the effectiveness of the new approach. Currently, FLT's employ a “one-approach-fits-all” strategy when locating features in different systems. The finding of this work implies, that it could be valuable for future research to characterize software systems towards picking more optimal FLT's for that system (or the most optimal configuration of such an FLT) to efficiently work with that particular type of the software.

1.9 Structure

This work is organized in the following manner:

• In the next chapter, Chapter 2 the related work is summarized and the research gap is identified. The chapter starts with the literature review protocol: existing taxonomies were reviewed and a systematic literature review protocol was used to update the review from 2011-2015. The reviewed material follows: a taxonomy of FLT's with examples for each category is presented, focusing on TFLT's and TFLT_{IR} in particular, discussion of the trends in FLT's going forward and how FLT's are evaluated. The chapter concludes with more specifically related work: the vocabulary paucity problem in TFLT's and how auxiliary material has been leveraged in the past to address it.

• In Chapter 3 the first contribution is discussed: a novel approach leveraging change-set descriptions is proposed - ACIR. The approach allows for the aggregation of change-sets for IR-based FL. First, the major steps of the approach are presented, followed by an in-depth description for each of these steps. Particularly, the difficulties and peculiarities of annotating source code entities using VCS's' change-set descriptions are described. The chapter concludes with an elaboration of the implementation details.
• Chapter 4 discusses the evaluation approach for ACIR: first presenting the methodology of the pilot study, followed by the revised methodological framework applied in the main study (the results of which are presented in Chapter 5). The methodology explains the design of experiment, the selection of subject systems and the criteria used for this selection, the benchmarking, the metrics used, and the replication procedures.

• Chapter 5 presents the evaluation of ACIR as per the pilot study and the large-scale experiment. At the beginning of the chapter, a small-scale pilot study evaluating ACIR is presented. Following the empirical observations of the pilot study, the scaled-up evaluation of ACIR was primarily oriented towards finding its best-performing configuration. The best-performing configuration was identified in terms of granularity, recentness (including aggregation of recent change-sets by change request), and filtering of non-functional “management” change-sets. The chapter provides descriptive data and statistical significance testing of the results. These are followed by the analysis of said results and critical discussion.

• Chapter 6 follows from the findings of the previous chapter of the best-performing configuration of ACIR. The best-performing configuration was compared against the TFLT\textsubscript{B}. A dichotomy in results was discovered with some systems reacting better to ACIR and some to TFLT\textsubscript{B}. Consequently the subject systems of the experiment were characterized with respect to the dichotomy discovered. This dichotomy finding also led to the proposed hybridization of the two approaches: the best-performing configuration of ACIR is used in conjunction with TFLT\textsubscript{B} approach to evaluate their effectiveness in combination. The results, their analysis, and discussion are presented in this chapter.

• Chapter 7 summarizes this work. The contributions of this work are revisited along with the major findings. The results of the findings are generalized to draw wider conclusions, implications for academic and industrial applications, and directions for future work. The threats to validity that could affect the findings of this work are also presented in this chapter.
2

Literature Review

The first part of this chapter concerns itself with the literature review protocol: in overview, well-known papers, describing FLTs en-masse were reviewed and then a literature search protocol was used to update the review to include papers produced up to, and including, 2015. The literature review was then expanded beyond the area of FL, but only to include the related techniques from BL, IA, and TLR, that leverage change-sets.

The identified material is reviewed in the second section. The section starts with a taxonomy of FLTs with concrete examples for each category. Next, the TFLTs and their underlying analysis are focused on. The section concludes with a discussion of how FLTs are evaluated, and highlights the trends in types of techniques going forward.

The third section funnels in to more specifically related work: the vocabulary paucity problem in TFLTs. The final section describes how auxiliary material has been leveraged in the past to address this problem.

2.1 Literature review process

The literature review process in this work was conducted in two phases, as shown in Figure 2.1. In the first phase, the set of the state-of-the-art FLTs (circa 2011) was identified by combining the FLTs reviewed by Dit et al. (2013a) and Rubin and Chechik (2013) and the list of those FLTs was expanded to include newer research. In the second phase, the state-of-the-art FL, BL, IA, and TLR
2.1 Literature review process

approaches, utilizing change-sets, were reviewed: the FLTIs, utilizing change-sets were picked from the set of phase 1 FLTIs and an additional literature search was conducted in phase 2 to identify the set of BL, IA, and TLR techniques, utilizing change-sets. Even though, during the research, these two phases detailed below happened some time apart from each other, they are presented here together for consistency.

The literature search elements of a systematic literature review were adopted in this review. A systematic literature review is a type of literature review, originating in medical research, that defines a rigorous methodology to answer research questions of that review (Dyba et al., 2005; Kitchenham et al., 2006; Petersen et al., 2008). Because of the rigorous methodology, this type of literature review has a number of advantages, such as reduced bias when selecting papers (Petersen et al., 2008), and, hence, has been suggested for researchers and practitioners in software engineering (Dyba et al., 2005; Kitchenham et al., 2006; Petersen et al., 2008).

2.1.1 Phase 1: Identifying the State-of-the-art FLTIs

In the first phase, the existing well-known literature, describing FLTIs en-masse, was examined (Dit et al., 2013a; Rubin and Chechik, 2013). These two review papers together cover high quality FLT research (as evidenced by their venue-selection criteria) up to February 2011. To assess the state-of-the-art in FL at the time of the review, this list of FLTIs was updated to include the work from January 2011 to April 2015, using the literature search elements of a systematic literature review, such as defining the search keywords and inclusion/exclusion criteria. The venues and the exclusion criteria, identified and used by Dit et al. (2013a) were utilized.

2.1.1.1 Existing Literature Reviews of Feature Location Techniques

Dit et al. (2013a) performed a systematic literature review to answer research questions about the type of analysis utilized by existing FLTIs, the trends in FLTIs according to their type of analysis, and the limitations in the evaluation
of existing FLTs. The authors manually searched for the relevant papers in 25 research venues. Their inclusion criteria would accept papers that:

- describe novel FLTs;
- evaluate existing FLTs;
- present tools that implement novel or existing FLTs.

The papers, that focused on improving the performance of underlying analysis (for example, IR model in textual analysis) but not the FL process per se, were excluded (exclusion criteria).

Following these criteria, they selected 89 research papers that have contributed to the subject of FLTs, starting from 1992 and going to February 2011. Of these 89 research papers, 60 presented novel or significantly modified approaches to FL. (The number of unique FLTs is less than the number of papers, because many
papers were follow-ups to the previous research by the same team of authors.) They systematically reviewed and classified these FLTs into a taxonomy using seven dimensions. In their taxonomy, the type of analysis used for FL is the main dimension (this defines the type of a FLT). There were five distinct types of FLTs: dynamic, static, textual, historical, and other. Also, there was a number of hybrid FLTs: various combinations of the five distinct types. According to their taxonomy, 10 research papers described purely dynamic techniques, eight research papers described static techniques, 12 research papers described textual techniques, 24 research papers described hybrid techniques, and six research papers described other techniques.

The survey by Rubin and Chechik (2013) focused on applying automated FLTs in the context of ad-hoc software product families and targeted helping practitioners decide on which FLT to use as part of locating features for systematic reuse in that context. Software product families are collections of software products, distinguished in terms of features (Kastner et al., 2014). The ad-hoc approach implies that a non-systematic, usually “clone-and-own” approach is used to create these software product families (Kastner et al., 2014; Rubin and Chechik, 2013). Because features are central to the software product families, feature location should assist as the primary step in the process facilitating their systematic reuse in moving from ad-hoc software product families to more formalized re-use.

Rubin and Chechik (2013) do not describe the process they employed in their literature search, but the authors still analysed 24 FLTs. According to the authors, 22 of these FLTs overlap with the FLTs identified by Dit et al. (2013a). However, the comparison of FLTs in these two reviews showed that there is only one FLT by Robillard and Dagenais (2008), described in Rubin and Chechik (2013) work, that is different. This approach by Robillard and Dagenais (2008) is more related to IA (find source code entities that should be modified as part of a change task) and because of that it was analysed with other IA techniques in phase 2.

Similar to Dit et al. (2013a), Rubin and Chechik (2013) provide a taxonomy of FLTs and classify FLTs according to the type of analysis used. However, there are just two types of analysis in their taxonomy: dynamic and static, each having
two subtypes, called guided and unguided. The number of dimensions in their
taxonomy is limited to 5.

In this PhD research reported on here, no additional literature search was
conducted to identify FLT papers, appearing in the literature prior to 2011. The
papers, identified in Dit et al. (2013a) and Rubin and Chechik (2013) reviews,
were used exclusively when reviewing FLTs prior to 2011 as:

- The systematic literature review processes, employed by the authors of these
  reviews were peer reviewed for rigour and, for the first review, this was done
  at the level of Institute for Scientific Information (ISI) journal.

- The authors explicitly used 25 top-quality, relevant, peer-reviewed venues
to conduct their search and employed explicit, relevant inclusion criteria to
select the papers. Such an approach would seem to objectively identify the
vast majority of high quality FLT papers.

- The overlap between the reviews: 60 papers, describing FLTs, were selected
  by Dit et al. (2013a). In comparison, Rubin and Chechik (2013) describe 24
  FLTs found in 27 papers and 23 of these FLTs overlap with FLTs, reviewed
  by Dit et al. (2013a).

- In terms of the period covered, the authors cover almost 20 years of FL
  research, up to February 2011.

The above reasons do not guarantee that all the relevant FLT literature was
identified by Dit et al. (2013a) and by Rubin and Chechik (2013) for the given
period. However, the rigorous literature search process, the high quality venues
selected, the number of FLTs identified, and the fact that these reviews cover
highly similar ground, in terms of FLTs identified for a given period, suggests
that the corpus of identified FLTs is a more-than-adequate representation of FL
research for that period, and that further analysis would only incrementally affect
the outcome for that period.

1In fact, it is enough to review FLTs, identified by Dit et al. (2013a) only, because they
fully overlap with the FLTs, identified by Rubin and Chechik (2013), except for one technique
mentioned earlier, that was reviewed in the second phase, along with IA papers.
2.1 Literature review process

However, the four years that have passed after the reviews suggest that the list of FL literature needs to be updated to include newer FLTs. To update the literature, a new literature search protocol was designed. The next section describes this new protocol and its differences from the one used by Dit et al. (2013a).

2.1.1.2 The Literature Search Protocol Used to Update The Existing FL Reviews

The literature search protocol, used in this phase, is different from the search protocol used by Dit et al. (2013a) in two major aspects:

- The search engines of well-known digital libraries were used to retrieve potentially relevant papers: an ease-of-review adaptation.
- The inclusion and exclusion criteria were amended.

2.1.1.2.1 Using Digital Libraries to Find FLT Papers

It seems that Dit et al. (2013a) adopted the following literature search strategy:

1. The list of potentially relevant venues was identified.

2. Every paper in those venues, falling in the period between 1992 and February 2011, was inspected, to assess its relevance to the systematic literature review.

Such an approach seems to be inefficient, because a potentially large set of irrelevant papers has to be assessed. In this work, the search engines of digital libraries were used to retrieve a set of potentially relevant papers, albeit from a wider range of venues (the search engines, as shown below provide coverage over the explicitly identified original venues reviewed by Dit et al. (2013a) and many other venues):

- ACM Digital Library\(^1\)

\(^1\)http://dl.acm.org/
2.1 Literature review process

- IEEEXplore Digital Library
- Google Scholar
- SpringerLink
- ScienceDirect
- Citeseer

These libraries provide access to the digital papers of the corresponding publisher (the owner of these libraries), or, in the case of Google Scholar, the library aggregates digital papers of many various publishers.

The following pseudo search string (the actual search string had to be modified with accordance to the syntax rules of each concrete search engine) was used with the search engines of the digital libraries above to find the relevant literature:

“feature location” OR “feature identification” OR “concept location” OR “concern location” OR “concern mapping” OR “concern identification”

The synonyms for feature location, used in this string, were picked by manually reviewing the keywords frequently used in FL literature, identified by Dit et al. (2013a). Some authors distinguish between features, concepts, and concerns (Marcus and Haiduc 2013) and the location activities associated with them (Scanniello and Marcus 2011). However, the differences in heuristics of such approaches seem to be negligible to the extent that they can be analysed and referred to en-masse as feature location (Dit et al. 2013a).

To further narrow down the literature search results, the following filters were applied when using the search engines of the digital libraries:

2.1 Literature review process

- Only the papers, published between January 2011 (inclusive) and April 2015 (the date of the review), were selected. Some overlap with the period used in the literature search by Dit et al. (2013a) (they searched up to February 2011) was left intentionally so that no potentially important paper is overlooked.

- Only the papers, published in the 17 venues, described below were selected. Dit et al. (2013a) analysed the literature from 25 venues, but ultimately, only 14 of these venues were selected for the literature search in this work. The selection was performed in three steps:

1. First, 14 venues were selected that contributed more than one paper to the review. Given the 19 years period, this suggests that the appearance of FLT literature in the remaining 11 venues was incidental rather than systematic.

2. Additionally, the “Empirical Software Engineering” journal was selected. It has contributed just one paper to the review by Dit et al. (2013a), but later search showed that the number of FLT papers has increased in 2011-2015 period (at least two FLT papers were found in that period).

3. Also, one conference (“International Workshop on Visualizing Software for Understanding and Analysis”) was removed from the initial list. It contributed two papers to the literature review by Dit et al. (2013a), however, the papers it contributed were mostly related to the visualization (as the conference title implies) of a system and were less relevant for this work.

The list of these initial 14 venues was further amended to include three more venues (not used by Dit et al. (2013a)) based on the analysis of the returned paper titles and topics of the associated venues, suggesting that these venues did concern themselves with feature-location type work: if they had greater than one paper on feature location. The complete list of 17 venues used in this work is shown in Table 2.1.

The table shows the changes made to the original list of the 14 venues, mentioned above. The “State” attribute describes the type of change made to the original list. If the value of the attribute is “Added”, the corresponding venue was
### 2.1 Literature review process

**Table 2.1:** The amended list of venues used in the current literature search

<table>
<thead>
<tr>
<th>Name</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Journals</strong></td>
<td></td>
</tr>
<tr>
<td>Journal on Software Maintenance and Evolution: Research and Practice</td>
<td></td>
</tr>
<tr>
<td>(Since 2012 - Journal of Software: Evolution and Process)</td>
<td></td>
</tr>
<tr>
<td>Journal on Systems and Software</td>
<td></td>
</tr>
<tr>
<td>ACM Transactions on Software Engineering and Methodology</td>
<td></td>
</tr>
<tr>
<td>IEEE Transactions on Software Engineering</td>
<td></td>
</tr>
<tr>
<td>Information and Software Technology</td>
<td>Added</td>
</tr>
<tr>
<td>Empirical Software Engineering</td>
<td></td>
</tr>
<tr>
<td><strong>Conferences</strong></td>
<td></td>
</tr>
<tr>
<td>Asia Pacific Software Engineering Conference</td>
<td></td>
</tr>
<tr>
<td>ACM International Conference on Aspect-Oriented Software Development</td>
<td></td>
</tr>
<tr>
<td>International Conference on Automated Software Engineering</td>
<td></td>
</tr>
<tr>
<td>European Conference on Software Maintenance and Reengineering</td>
<td></td>
</tr>
<tr>
<td>European Software Engineering Conference/ACM SIGSOFT</td>
<td></td>
</tr>
<tr>
<td>Symposium on the Foundations of Software Engineering</td>
<td></td>
</tr>
<tr>
<td>International Conference on Software Engineering</td>
<td></td>
</tr>
<tr>
<td>International Conference on Software Maintenance (Since 2014)</td>
<td></td>
</tr>
<tr>
<td>International Conference on Software Maintenance and Evolution</td>
<td></td>
</tr>
<tr>
<td>International Conference on Program Comprehension</td>
<td></td>
</tr>
<tr>
<td>Working Conference on Reverse Engineering</td>
<td></td>
</tr>
<tr>
<td>Mining Software Repositories</td>
<td>Added</td>
</tr>
<tr>
<td>Source Code Analysis and Manipulation Working Conference</td>
<td>Added</td>
</tr>
<tr>
<td>International Workshop on Visualizing Software</td>
<td>Removed</td>
</tr>
<tr>
<td>for Understanding and Analysis</td>
<td></td>
</tr>
</tbody>
</table>

not part of the original list of the 14 venues, but is used in the current literature search. If the value of the attribute is “Removed”, the corresponding venue was part of the original list of the 14 venues, but is not used in the current literature search. If no value is provided, the corresponding venue remained unchanged in the list.

To summarize, the approach, described in this section, allows the use of search engines of digital libraries to find potentially relevant papers, to eliminate the
2.1 Literature review process

need for manual inspection of all the literature in the venues. However, such an approach is sensitive to the input keywords, that are used in the search string. These keywords need to be picked carefully to select the relevant literature.

2.1.1.2.2 The Inclusion and Exclusion Criteria for FLT Papers  Because the goal of this literature search was to identify the state-of-the-art in FLTs, some of the criteria defined by Dit et al. (2013a) (see Section 2.1.1.1), such as inclusion of case studies of existing FLTs and implementation papers of existing FLTs, seemed to be inappropriate. Hence, the inclusion and exclusion criteria were modified to meet the goal of this review.

Therefore, one inclusion criterion and two exclusion criteria were used. The inclusion criterion accepts papers, presenting a novel FLT, or a significant modification to an existing FLT.

The first exclusion criteria rejects papers that do not provide evaluation of the FLT. The presence of evaluation indicates that a FLT was implemented at least as a proof-of-concept and is not merely an idea. It could happen that after presenting the abstract idea, researchers find the fundamental flaws with this idea and decide to abandon this idea. Hence, it seems reasonable to exclude papers at such an early research stage. For example, the paper by Haiduc (2011) was retrieved by the search engine, because it matched the “concept location” keyword. This paper described a potential novel FLT, but didn’t provide any evaluation.

The second exclusion criteria was borrowed from Dit et al. (2013a), as mentioned in Section 2.1.1.1 the exclusion of techniques that improve the performance of FLTs, but not the FL process per se (e.g. improve the speed of a FLT, but not the effectiveness).

An example of a paper that passed the inclusion criteria and exclusion criteria was that of Petrenko and Rajlich (2013). This was retrieved by the search engine, because it matched the “concept location” keyword. This paper presented a novel FLT, assessed that novel FLT, and it was not a performance improvement paper. Hence, this paper was included in the literature review.
2.1 Literature review process

2.1.2 Phase 2: Identifying Approaches that Leverage Change-sets

The literature review in phase 1 identified a potential vocabulary paucity problem in TFLT_{IRs}. Leveraging change-set descriptions could potentially address that problem. However, to get a broader picture and to be sure that all the relevant work was reviewed, towards raising confidence that this was a novel and worthy area of research, the literature review in phase 2 was expanded beyond FL, to include work from related areas such as BL, IA, and TLR that leveraged change-sets.

2.1.2.1 The Areas Related to FL: BL, IA, and TLR

Here, the overlap between FLT_s and BL, IA, and TLR areas is visited in detail to explain their inclusion in the literature review.

- Bug localization is associated with finding relevant source code entities as part of corrective software maintenance \( \text{(Lukins et al., 2008)} \). The major difference between FL and BL is that a BL targets only bug reports. As user’s bug reports typically focus on the functionality that does not work as intended or needs enhancing \( \text{(Lientz and Swanson, 1980)} \), the developer’s task is to move from the function-based report to the associated source code. Hence the difference is more conceptual than methodological to the extent that both methods might be used interchangeably with only the input being adjusted. Indeed, in the literature BL is sometimes used as a synonym of FL \( \text{(Ali et al., 2012; Sisman and Kak, 2013; Wang and Lo, 2014)} \).

- IA scrutinizes the effect of introducing a change into a current state of a software system \( \text{(Bohner and Arnold, 1996)} \). The change usually starts as an intention to add or alter existing functionality and is associated with a change request, which describes the initial intent in terms applicable to the software. After a developer modifies the source code a number of side effects may happen, that might provoke erroneous behaviour of a system. In other words, any change to a software system can cause ripple-effects to happen. IA is expected to mitigate the effect by recognizing the prone source code
2.1 Literature review process

entities, that are required to be altered (the source code entities may not necessarily belong to the same feature, but could be from tightly coupled features). Impact analysis is usually performed after the entry point to the feature was identified in the code, but again can be conceptualized as finding locations in source code: this time the locations sought are based on code changed rather than a functional query. It is an integral part of an incremental change process (Rajlich and Gosavi, 2004). Impact analysis is an actively studied field and a number of techniques were designed (Canfora and Cerulo, 2005; Gethers et al., 2011; Zanjani et al., 2014), many similar to FLTs.

- TLR allows for connecting system requirements, which might be in a form of documents, use cases, or other free text form, and the source code where they are implemented (Ali et al., 2013a; Antoniol et al., 2002; Diaz et al., 2013). Many of these requirement documents are phrased in terms of functional requirements and so, generating this trace information retrospectively, can be considered a form of feature location.

Though there exist differences between FLTs and BL, IA, and TLR techniques, sometimes these differences are quite subtle. The BL approaches seem to be close heuristically and the application domain overlaps to a large extent. IA usually identifies the full feature extent, but may also identify other source code entities that should be changed because of a “ripple effect”. TLR could be thought of as a FL that is applied repetitively to a software system until all the documentation (or other abstract feature descriptions) is mapped to source code.

Because of these similarities, the literature review includes BL, IA, and TLR approaches along with the FLTs.

2.1.2.2 The Literature Search Protocol for Relevant BL, IA, and TLR Techniques

For this part, the literature search protocol similar to the one described earlier in Section 2.1.2.1 was used: the same digital libraries (and their search engines) and the same venues, as described in Section 2.1.1.2.1 were used. There were
a few differences though. The search string was modified to suit the purpose of this literature search. This pseudo search string is shown below:

(“bug location” OR “bug localization” OR “impact analysis” OR “traceability link recovery” OR “requirements traceability” OR “traceability links”) AND (“software repositories” OR “version histories”)

Also the time period was amended: there does not seem to be a large systematic literature review for these research areas, similar to the review by Dit et al. (2013a) for FL. Hence, the time period was extended to cover the last 10 years of research: papers older than 10 years (before 2005) were excluded from the review, because BL, IA, and TLR are actively studied areas and therefore it is unlikely that state-of-the-art papers in these areas that considered change repositories are older than 10 years.

The following inclusion criteria applied:

• The paper has to describe a novel BL, IA, or TLR technique or a significant modification to an existing technique.

• The technique, presented in the paper, has to leverage change-sets.

The exclusion criteria are the same as described in Section 2.1.1.2.2 and would reject the papers that contribute to performance improvements or that do not assess the approach.

2.2 Overview of FLT

The literature search in phase1, described in Section 2.1.1 identified 87 relevant FL papers. Of these 87, 60 papers come from the review by Dit et al. (2013a) and 27 were identified as part of the literature update (see Section 2.1.1.2). Because the body of literature is abundant, it is convenient to describe the FLT in a systematic way using taxonomies. The taxonomies have been used before in FL research to describe FLT (Dit et al. 2013a, Rubin and Chechik 2013) and were
introduced in Section 2.1.1.1. These taxonomies are presented in more detail in the following section.

2.2.1 The Existing Taxonomies of FLTs

As mentioned in Section 1.3, the type of analysis used in a FLT is often used to classify the FLT. 

Dit et al. (2013a) divide FLTs into dynamic, static, textual, historical, other and into various combinations of these types (hybrid approaches). According to the authors, dynamic FLTs rely on the execution traces recorded during system runtime. Static FLTs examine software system’s structural dependencies. Textual FLTs utilize text available in source code. Also, the authors discussed the historical FLTs’ group, which borrows from the idea of textual FLTs, but utilizes a distinctly historical (e.g. VCS) dataset.

Rubin and Chechik (2013) distinguish between dynamic and static FLTs, but also introduce subtypes for each group. According to them, the static and dynamic groups could be further divided into plain and guided. The plain and guided subtypes define, whether additional details and rankings are provided (guided) or an unsorted set of results is returned (plain).

2.2.2 The Taxonomy of FLTs Used in This Work

In this work, a refined taxonomy is used to classify FLTs. The main purpose of this taxonomy is to focus on the FLTs that use non-source code textual analysis (historical according to Dit et al. [2013a]) and also to assist in describing the FLTs in this review in a systematic way. In essence, the classification of FLTs used in this work combines the taxonomies of Dit et al. (2013a) and Rubin and Chechik (2013) and re-factors the static/textual categories of Dit et al. (2013a), towards a more hierarchical structuring (see Figure 2.2 for comparison):

- The static type has two subgroups: structural and textual.
- The textual group has two subgroups: source code and meta-data.
2.2 Overview of FLTs

This latter distinction is the most important part of the taxonomy used in this work. The source code group utilizes source code data, whereas the “meta-data” group utilizes non source code sources. The main reason for this re-factoring is that existing taxonomies make it hard to see the FLTs that use non source code textual sources. For example, in the taxonomy by Rubin and Chechik (2013) such FLTs are part of the broad static group, which makes them hard to discern. Dit et al. (2013a) have a historical type group for a subset of such approaches. However, none of the FLTs, that had historical-analysis elements were assigned to this group. For example, the FLTs by Chen et al. (2001); Cubranic and Murphy (2003); Ratanotayanon et al. (2010b), despite their truly non-source code nature, were classified as “other” (a bucket category) and the FLT by Cleary and Exton (2007) was classified as “textual”, thus making it hidden in the broad textual group. Instead, the textual meta-data group in this taxonomy allows a specific category for identification of the amount of TFLT work using non source code data sets.

Another problem that the proposed taxonomy addresses is the ambiguity in the naming that is used to refer to FLTs in the other taxonomy. For example, static FLTs in the taxonomy by Rubin and Chechik (2013) are the techniques utilizing any statically available information (structure and text in source code), whereas static FLTs in the taxonomy by Dit et al. (2013a) refer to the techniques utilizing structural information only. In this work, the static group encapsulates all the techniques utilizing any statically available information, similarly to Rubin and Chechik (2013). However, unlike the latter authors, this group has two explicit subgroups: structural and textual, making explicit the Dit et al. (2013a) types, but within the static category.

The guided and plain subgroups used by Rubin and Chechik (2013) are not used in this taxonomy because there is no need to distinguish between FLTs that generate unsorted (plain) and sorted (guided) output.

Finally, similarly to Dit et al. (2013a), dynamic FLTs and various types of hybrid FLTs are included in the taxonomy. The group of the other FLTs, those that do not utilize the runtime or static content of a software system, is represented by only one such FLT by Robillard and Murphy (2003). It utilized the interactions of a developer with source code for feature location.
2.2 Overview of FLTs

2.2.2.1 Populating the Taxonomy of FLTs

The papers identified as part of the phase 1 literature search (Section 2.1.1) were added to the taxonomy and the classification results are summarized in Table 2.2. In the table, the numbers for each group signify the number of FLT papers that fall into a particular category. The numbers inside the parentheses indicate the number of unique FLTs, described in those papers. The numbers are different, because approaches can appear in several papers by the same group of authors that, for example, extend the approach/empirical evaluation\(^1\). For the complete

\(^1\)Michael Würsch has authored two papers: one in 2010 (identified by Dit et al. (2013a)) and one in 2013 (identified in the literature update), that describes one unique approach (textual (source code + meta-data)). This unique approach was counted only in the current literature review to avoid duplicates.
2.2 Overview of FLTs

list of papers see Appendix A: Listing of All FL Papers Used in Taxonomic Structure in Phase 1

Table 2.2: The statistics of FL papers according to the classification used in this work.

<table>
<thead>
<tr>
<th>Type of FLT</th>
<th>Papers from Dit et al. &amp; Rubin and Chechik</th>
<th>Papers from current literature search</th>
<th># Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamic</td>
<td>10 (8)</td>
<td>3 (3)</td>
<td>13 (11)</td>
</tr>
<tr>
<td>Structural</td>
<td>8 (4)</td>
<td>0 (0)</td>
<td>8 (4)</td>
</tr>
<tr>
<td>Textual (Source Code)</td>
<td>9 (8)</td>
<td>8 (7)</td>
<td>17 (15)</td>
</tr>
<tr>
<td>Textual (Meta-Data)</td>
<td>1 (1)</td>
<td>0 (0)</td>
<td>1 (1)</td>
</tr>
<tr>
<td>Other</td>
<td>1 (1)</td>
<td>0 (0)</td>
<td>1 (1)</td>
</tr>
<tr>
<td>Hybrid: Dynamic Structural</td>
<td>9 (5)</td>
<td>2 (2)</td>
<td>11 (7)</td>
</tr>
<tr>
<td>Hybrid: Dynamic Textual (Source Code)</td>
<td>6 (5)</td>
<td>1 (1)</td>
<td>7 (6)</td>
</tr>
<tr>
<td>Hybrid: Hybrid Structural Textual (Source Code)</td>
<td>1 (1)</td>
<td>1 (1)</td>
<td>2 (2)</td>
</tr>
<tr>
<td>Hybrid: Structural Textual (Source Code)</td>
<td>8 (6)</td>
<td>9 (8)</td>
<td>17 (14)</td>
</tr>
<tr>
<td>Hybrid: Textual (Source Code + Meta-data)</td>
<td>6 (2)</td>
<td>3 (3)</td>
<td>9 (5)</td>
</tr>
<tr>
<td>Hybrid: Structural Textual (Source Code + Meta-data)</td>
<td>1 (1)</td>
<td>0 (0)</td>
<td>1 (1)</td>
</tr>
<tr>
<td># Total</td>
<td>60 (42)</td>
<td>27 (25)</td>
<td>87 (67)</td>
</tr>
</tbody>
</table>

In the rest of this work, the classification of FLTs, as defined in this new taxonomy is used, if not stated otherwise. In the following sections, the overview of the various types of FLTs is given.

2.2.2.2 Dynamic FLTs

In dynamic FLTs, scenarios are used to run a software system in such a way that certain features of the software system are/are not triggered as part of the software system’s execution. These scenarios are usually test or use cases, and involve various input types: for example, external files or parameters of a software system. Before the execution, the software system is instrumented in such a way that runtime information can be collected during or after the execution is finished. This runtime information is called execution trace information. The execution traces store an ordered set of source code entities (often these are sub-program calls) as they appear during the software system’s execution. Therefore, for each scenario exercised there is a corresponding execution trace and, hence, the features of that scenario can be mapped to the source code entities from the execution trace. However, only a small fraction of source code entities in
2.2 Overview of FLTs

Execution traces are relevant to a feature \cite{Wilde and Scully 1995}. Dynamic FLTs employ different strategies to identify that fraction.

One of the first dynamic FLTs was introduced by \cite{Wilde and Scully 1995}. The authors implemented scenarios as test cases targeted at specific features for capturing relevant sets of source code entities. According to the authors, there exists a set of shared (e.g. utilities) source code entities (executed in several scenarios), a set of mandatory entities (executed in all scenarios that exercise a feature), a set of feature-relevant entities (executed in at least one scenario that exercises a feature), and a set of feature-unique entities (which are only exercised by the scenarios exercising that feature). To identify the feature-unique source code entities, the authors created 252 test cases: half of those trigger a feature and the other half do not. The source code entities are feature-unique if they appear in the execution traces of the test cases that turn the feature on, but do not appear in the execution traces of any of the test cases that turn the feature off. The set of 252 test cases seems to be very large, and after further experiments it was concluded that only 12 test cases are needed on average to produce the same result (the set of feature-unique source code entities). The authors state that these 12 test cases could be reduced even further, down to just two pairs of test cases, if system experts are involved when creating these test cases. But this is the caveat to their approach: the generation of judicious test-cases involves careful, expert consideration.

In a more recent study, \cite{Yousefi and Sartipi 2011} used a dynamic FLT to locate features in service oriented architectures (SOA). In SOA, services (software systems) could be implemented in a variety of languages, could run on a variety of platforms, and communicate with each other and the end-user via messaging. Such an architecture presents additional difficulty to dynamic FLTs, because the execution could be asynchronous, distributed and concurrent. This means that, scenario-triggered, feature-related code can run in several software systems at the same time and interleave with other scenarios currently running. The main contribution of \cite{Yousefi and Sartipi 2011} was to aggregate execution traces of various software systems related to a specific feature and so can be seen as an extension of \cite{Wilde and Scully 1995} work for more modern software systems.
In another study, Heydarnoori et al. (2012) applied dynamic FLT to locate features in the *application programming interface* (API) of a framework. In computer science, a framework is a domain specific environment that facilitates the creation of domain specific software systems. The frameworks usually provide templates, tools, and source code entities that are accessible through the framework’s API. Usually frameworks control the execution of a software system, which marks them out as different from more traditional libraries (collections of source code entities). The authors used programs, that leverage framework’s API, as execution scenarios. Unlike Wilde and Scully (1995), to identify feature-unique source code entities in execution traces, they marked the beginning and the end of feature related API calls in the programs as their traces.

In general, the application of dynamic analysis to FLTs showed that this type of analysis can be effective when locating features in source code. Arguably, the major advantage of such analysis is that, given a scenario that can turn on/off a feature, one is guaranteed to find at least a fraction of feature-related source code entities. However, there are also fundamental problems with dynamic FLTs:

- The core problem with dynamic FLTs is their inability to locate features that are always executed. In other words, there doesn’t exist a scenario that would turn such a feature off. Thus source code entities that belong to this feature will always be part of the execution trace.

- The dynamic FLTs seem to be good at locating entry points to a feature, but not the extent (see Section 1.1) of a feature (Wilde and Scully, 1995).

- Though some work has been done on reusing existing test cases for dynamic FLTs (Ziftci and Kruger, 2012), the scenarios still need to be created and maintained as the software system evolves. (This is probably less of an issue for feature location in legacy code, where feature location is likely a “once-off” activity.) The FLTs are supposed to reduce the effort of feature location in source code, as discussed in Section 1.2. The creation and maintenance of scenarios, which is a necessary part to dynamic FLTs, requires additional expert effort, raising the question: how useful are dynamic FLTs in reducing the FL effort?
2.2 Overview of FLTs

- The software systems need to be executed to collect execution traces. This could be a concern for software systems with long execution times. The software system has to be executed at least twice for each feature, to collect the execution traces with and without specific feature-related source code entities. However, [Wilde and Scully (1995)] suggest that 12 test cases are needed on average. Hence, the execution time could easily go up to 12 times respectively.

Further information on dynamic FLTs could be found in the [Dit et al. (2013a)] review.

2.2.2.3 Static Structural FLTs

Static structural FLTs analyse structural dependencies of source code entities. Since, source code entities don’t often operate in isolation, but instead communicate with each other and pass information, their interactions could be leveraged to support FL. For example, a developer could start at a source code entity he/she knows is part of a feature and inspect all outgoing sub-program calls, related to that entity.

[Chen and Rajlich (2000)] were among the first to apply static structural analysis to FLTs using an “abstract static dependency graph”. The graph might represent either control or data flow of a software system. In the first case, the nodes of a graph are sub-programs and the edges are the sub-program calls. In the second case, the nodes are variables and the edges correspond to data transfer. When using the graph to locate the features, a developer has to decide on a starting point (i.e. graph entry point). In this case he might either start from the “main” sub-program, from any arbitrary node in the graph, or, for example, use information from a change request to reason about the starting point. The static techniques would then further assist a programmer as he/she walks the graph. Usually a programmer has to make traversal decisions. But the graph itself means that these decisions are a subset of all the navigation decisions they could have made and techniques would assist by suggesting unexplored paths and showing already selected elements. A developer might choose between leveraging node names, a breadth-first, depth-first search strategy. They might also decide
2.2 Overview of FLTs

if bottom-up or top-down feature location traversal is preferable \cite{Chen2000}. For example, a change request indicates that a sub-program $A$ could be a good starting point. $A$ calls two other sub-programs $B$ and $C$. A developer inspects the latter two sub-programs and concludes that $B$ is relevant, whereas $C$ is not. He would then continue to inspect the neighbourhood of $B$ in a similar manner.

While the approach by \cite{Chen2000} facilitates the graph traversal by offering the nodes for inspection and remembering the visited nodes, it still requires much developer’s interaction. For example, a developer has to manually inspect all the nodes that are connected to the starting node and to decide on their relevance. A more advanced approach by \cite{Robillard2008} suggests the more relevant nodes for inspection by ranking them higher. The ranking is based on two metrics: “specificity” and “reinforcement”. For example, the sub-program is more specific if it called by a fewer number of other sub-programs. The sub-program is reinforced if it is called by other sub-programs that were already identified as part of a feature.

Since static structural FLTs usually require regular user interaction, they are mostly utilized as helper tools or in a combination with other types of FLTs \cite{Petrenko2013,Scanniello2011}. Static FLTs are useful when finding the extent of a feature or the set of source code entities as part of IA. The main disadvantage is the need for the starting point to be selected accurately. Starting from the “main” sub-program could be justified in the case of a program. However, large software system would accordingly have a large graph of entities and traversing all the nodes to locate the relevant ones would likely require significant effort.

Unsurprisingly then, the review of the most recent FLTs (from 2011 onwards) did not reveal any standalone structural technique (see Table 2.2). Most of the structural information was utilized in combination with other approaches. Other interesting reading on static structural FL could be found in the \cite{Dit2013} review and the \cite{Marcus2005} case study.
2.2.2.4 Hybridization of Dynamic and Structural FLTs

As mentioned in Section 2.2.2.2 and Section 2.2.2.3, pure dynamic and structural FLTs have their own disadvantages. For example, dynamic FLTs are good in identifying the starting points of a feature, but not the extent of a feature. Structural FLTs seem to be better at identifying the extent of a feature, but rely on the accurate starting points. Hence, it seems reasonable to mix these two approaches to FL, to compensate for their disadvantages.

A good example of a dynamic-structural approach is presented by Eisenbarth et al. (2003). The technique comprises both dynamic and static structural approaches and is implemented as a multi-phase model guided through interaction with a developer. In the first phase a domain expert creates scenarios that should trigger interesting functionality. The execution traces are recorded and reviewed by a user who would interpret them by means of formal concept analysis (FCA). FCA is an analysis technique that allows the inference of a hierarchical concept lattice from the relations of objects and attributes (Ganter and Wille, 1999). Eisenbarth et al. (2003) use FCA to derive feature to source code entities mappings and the generation of the concept lattice allows the user understand the set: sub-set relationship between features, in terms of their executed entities. The mapping obtained is the subject of a further static investigation conducted by a system analyst. The analyst examines the structural dependencies of feature related source code entities to include entities that could be a part of a feature’s extent.

2.2.3 Static Textual FLTs

Because the group of TFLTs is the largest group in the taxonomy (see Table 2.2) and because it is the focus of this work, this group is reviewed here in its separate section.

TFLTs leverage lexical information available directly in source code (textual source code group) or in auxiliary sources associated with a software system (textual meta-data group).

The input to TFLTs is a search query. The search query may consist of search keywords or it may also be a natural language sentence. The search keywords
or the words for the natural language sentence are derived from textual feature
descriptions. It seems that the textual feature descriptions usually come from
(Dit et al., 2013a; Marcus and Haiduc, 2013):

- Documentation of a software system;
- Change requests;
- Domain knowledge of a system expert (the expert knows how to describe a
  feature);
- Other textual sources (e.g. team communication textual documents such as
e-mails).

Given a search query, the task of a TFLT is to return the list of source code
entities that match the query. Broadly speaking, to accomplish that, the TFLTs
usually have to:

1. Represent the source code entities using the textual information in that
   source code or meta-data that has been associated with source code. For
   example, comments and variable names could be used to represent sub-
   programs.

2. Match the search query against these textual representations of source code
   entities.

Meaningful textual data in source code usually comes from:

- Identifiers: these are the names of source code entities, such as variables’
  names, sub-programs’ names, classes’ names, packages’ names, interfaces’
  names, and other.

- Comments: this is a natural language text, that describes the source code.

- Literals: a fixed (usually string) value in source code.

The textual data in meta-data is more varied. The most commonly used are:

- The textual data in change-sets of VCSs: the data could be the author’s
  name, time-stamp, and/or descriptions.
2.2 Overview of FLTs

- The textual data in change requests of ITSs: the data usually comes from the summary and description of a change request.

The methods, used in TFLTs to match the search query, might be as simple as PM, or might utilize more sophisticated IR and NLP approaches (Dit et al., 2013a).

2.2.3.1 Pattern Matching in TFLTs

Pattern matching takes a regular expression and a set of strings as an input and returns a list of strings that match that regular expression. A regular expression is a string pattern that allows specification of the type of characters, the groups of characters, and the number of occurrences for groups or characters. For example, consider the following (regular expression) pattern and three strings $A$, $B$, and $C$:

<table>
<thead>
<tr>
<th>Pattern</th>
<th>$[a-z]{3}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A$</td>
<td>cat</td>
</tr>
<tr>
<td>$B$</td>
<td>896b</td>
</tr>
<tr>
<td>$C$</td>
<td>?+ =</td>
</tr>
</tbody>
</table>

Only the first string $A$ will be a match for the pattern, because it contains a sequence of three small case letters.

A more advanced usage of pattern matching could involve the specification of wild-cards or a pair (and some programs allow more) of regular expressions. The logical “OR” and “AND” operators could be applied to those regular expressions to match the strings that either match both of those expressions or just any one of them.

A very well-known implementation of PM is the Unix “grep”1 program.

Grep-like facilities were used to some extent in FLTs to support pattern matching (Chen et al., 2001). However, mostly they were used as a comparison against novel FLTs (Dit et al., 2013a).

PM is a fairly simple, usually readily available, and quick approach. However, this approach has two major drawbacks when applied to FL:

- It cannot rank the results.

2.2 Overview of FLTs

• One, very fine level of granularity of source code entities is supported: individual tokens.

Suppose a search query matches tokens in many lines of code. Because of the drawbacks of PM, a developer will have to inspect a potentially very large list of lines of code [Jordan et al., 2015].

2.2.3.2 Natural Language Processing in TFLTs

Natural language processing is the study of the interaction between natural languages and computers in terms of understanding and generation of natural language by computers.

The application of NLP to FL can be demonstrated by the work of [Hill et al., 2009]. The authors utilized NLP to tag the parts of identifiers in source code with parts of speech. For example, method (sub-program) identifiers in Java are known to follow certain name-structuring rules. Usually these rules imply that certain combinations of verbs, nouns, and prepositions are used. For example:

• Verb + noun: setRadius;
• Noun + preposition: instanceof;
• Verb + preposition: compareTo;
• Verb + noun + preposition: setArrayTo.

The technique by [Hill et al., 2009] used speech tags to organize the parts of identifiers into a hierarchy. In this hierarchy every part of an identifier (of certain part of speech) points to other parts of identifiers (of different parts of speech). In the example above, there are two methods “setRadius” and “setArrayTo”. The name of the first method is a combination of verb + noun, and the name of the second method is a combination of verb + noun + preposition. In the hierarchy, the word “set” would point to “radius” and to “array”. The “array” would point to “to”. Hence, if a search query contains just one search keyword “set”, the technique would suggest to expand this query using the keywords “radius” and “array”. This technique is potentially useful if a developer struggles to formulate a meaningful search query.
According to Dit et al. (2013a), the major limitation of NLP TFLTs is the added cost of computation. Indeed, these approaches usually require additional morphological parsing of textual data to identify parts of speech. However, it is unclear if this argument still fully holds going forward, given increasing computational powers.

A more interesting question is if it is necessary to apply NLP for FL. For example, it seems that an approach similar to Hill et al. (2009) could similarly suggest query keywords by recording co-occurrences of words in the identifiers in source code, disregarding their part of speech. The NLP part is unnecessary for such an approach. Also, it is unclear how this type of TFLTs compares to PM and TFLT_{IR}. To date, there does not seem to be a study comparing NLP TFLTs to other approaches (Hill et al., 2009, 2014a; Würsch et al., 2013). This preliminary state of assessment does not help in the evaluation of reason about the effectiveness of such approaches.

2.2.3.3 Information Retrieval in TFLTs

Probably the most largely studied group of TFLTs are TFLT_{IR}. The distinguishing characteristic of these approaches is that the textual data is treated as a “bag-of-words”: the order and semantic relationships between words are not important, but the multiplicity of words is important. For instance, PM is not a bag-of-words approach, because in PM multiplicity is not leveraged. NLP is not a bag-of-words, because in NLP the order and semantic relationships do matter.

In TFLT_{IR}, source code entities are represented by textual documents. The search queries are matched against these textual documents. In this section the most commonly used IR models underpinning this matching are briefly presented.

IR models can represent a corpus of documents as a matrix $A$ that has $M \times N$ dimensions. The $N$ is the number of documents in the corpus and the $M$ is the number of distinct terms (words) in the corpus of documents. Consider the following five documents:
2.2 Overview of FLTs

There are 11 distinct terms in these documents after the prepositions and frequent English words were removed. These terms, in lower case, are: “hoover”, “brush”, “animal”, “hair”, “pet”, “collect”, “fur”, “trade”, “common”, “alaska”, “state”. In this example, \( A \) is a \( 11 \times 5 \) matrix. It is impractical and inconvenient for mathematical computations to store the terms as strings. Therefore, the terms are converted into a numeric form.

For each term in a document, a numeric score is calculated. A common approach to scoring is to combine the statistics of the term frequency within the document and across the documents in the corpus. This statistic is called term frequency - inverse document frequency (TF-IDF) [Salton and McGill, 1983]. One common variant of TF-IDF formula is presented below.

\[
TF - IDF(t, d, D) = tf(t, d) \times idf(t, D)
\]  

where 
\[
idf(t, D) = \log \frac{N}{1 + |d \in D : t \in d|}
\]

In Formula (2.1) \( t \) is a term in a document \( d \) in a corpus of documents \( D \). In this example, the \( tf(t, d) \) function counts the number of times the term \( t \) appears in a document \( d \). The \( idf(t, D) \) function (see Formula (2.2)) counts how many documents in the corpus \( D \) contain the term \( t \), where \( N \) is the number of documents. For example, the TF-IDF score for the term “hoover”, in document \( d1 \) from the set of documents above, is calculated as follows: the \( tf \) of “hoover” equals 1, the \( idf \) equals \( \log \frac{5}{1 + 2} = 0.2218 \), and the TF-IDF is equal to 0.2218. Applying the TF-IDF (or other scoring function) to all the terms in all the documents in the corpus will transform the documents into numeric vectors in matrix \( A_{M,N} \), as shown in Table 2.3.

Such representation of documents in a corpus is called a vector space model (VSM). Because every document in such model is a numeric vector, the similarity
2.2 Overview of FLTs

Table 2.3: The VSM matrix of the five example documents.

<table>
<thead>
<tr>
<th></th>
<th>d1</th>
<th>d2</th>
<th>d3</th>
<th>d4</th>
<th>d5</th>
</tr>
</thead>
<tbody>
<tr>
<td>hoover</td>
<td>0.22</td>
<td>0</td>
<td>0.22</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>brush</td>
<td>0.22</td>
<td>0.22</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>animal</td>
<td>0.4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>hair</td>
<td>0.4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>pet</td>
<td>0</td>
<td>0.22</td>
<td>0.22</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>collect</td>
<td>0</td>
<td>0</td>
<td>0.4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>fur</td>
<td>0</td>
<td>0</td>
<td>0.22</td>
<td>0.22</td>
<td>0</td>
</tr>
<tr>
<td>trade</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.4</td>
<td>0</td>
</tr>
<tr>
<td>common</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.4</td>
<td>0</td>
</tr>
<tr>
<td>alaska</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.22</td>
<td>0.22</td>
</tr>
<tr>
<td>state</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.4</td>
</tr>
</tbody>
</table>

Between two such vectors could be calculated as the angle between them. Usually the cosine of the angle is calculated and hence such similarity is often referred to as a cosine similarity. The Formula 2.3 shows how to calculate the cosine similarity between a query vector \( q \) and document vector \( d \):

\[
\cos \theta(d, q) = \frac{d \cdot q}{\|d\| \times \|q\|}
\]  

(2.3)

For example, if the two vectors are orthogonal their cosine is equal 0, whereas if the vectors are equal their cosine is equal to 1. All other cases of partial similarity will have values between 0 and 1. When a search query is used to find the documents in the corpus, it is first converted into a numeric vector using the techniques described above. Then it is compared to all the other vectors in the matrix \( A \) using cosine similarity.

The above approach requires at least a partial match between the search query terms and documents’ terms (for convenience, such IR approaches are called VSM in the rest of the text). Hence, in VSM, not all potentially relevant documents can be retrieved. Consider the five documents above and the search query of two terms “pet” and “fur”. The document \( d_3 \), followed by documents \( d_4 \) and \( d_2 \), will match this query. However, it seems that document \( d_1 \) might be more relevant to this query than document \( d_3 \), despite the match of the latter and the search
query. Looking at the documents, one could conclude that they belong to two
different topics: documents $d_1$, $d_2$, and $d_3$ seem to relate to cleaning the pet fur,
whereas the documents $d_4$ and $d_5$ are related to the Alaska state.

Latent semantic indexing (LSI) technique is supposed to identify such hidden
concepts in the corpus of textual documents (Deerwester et al., 1990). The tech-
nique implies that the terms and documents in the corpus are related to these
hidden concepts. The matrix $A_{M,N}$, the VSM representation of $M$ terms and $N$
documents, is the input to this technique. The technique then decomposes
the matrix $A$ into three matrices $T$, $S$, and $D$ using the singular value decomposi-
tion (SVD). (The explanation of the rationale and matrix computations involved in
SVD is outside of the scope of this work. Please refer to Deerwester et al. (1990)
for further details.):

\[
A_{M,N} = T_{M,K} \times S_{K,K} \times D_{K,N}
\] (2.4)

In Formula 2.4, $K$ is the number of hidden concepts. The matrix $T$ stores
the information of how relevant each term $t \in T$ is to each concept $k \in K$. The
matrix $D$ stores the information of how relevant each document $d \in D$ is to each
concept $k \in K$. The matrix $S$ is a diagonal matrix (a matrix that has values
other than zeros only in its diagonal) of singular values (singular values are square
roots of eigenvalues, which, in turn, are the “characteristic” scalars of a matrix
(Deerwester et al., 1990)). The role of the diagonal matrix $S$ is to multiply its
values with the values of terms and documents in matrices $T$ and $D$ respectively,
thus scaling the latter with respect to concepts. The search query keywords are
then “matched” against the concepts, and the documents that are “closer” to
these concepts are returned.

Latent Dirichlet allocation (LDA) is another IR model that was recently used
in TFLT_{IR}s. LDA is a generative IR model: it generates the documents from
a set of topics $Z$ (every document is represented by a topic distribution) and
every word in a document is generated from topic $z \in Z$ ($z$ is represented by a
word distribution). There are two smoothing parameters $\alpha$ and $\beta$ that influence
the distribution of topics per document and words per topic respectively. Recall
the set of example documents used in this section. Suppose there are three
topics: “cleaning”, “pet”, and “alaska”. Then the distribution of topics in these documents could look as follows (note that the distribution numbers for topics in documents and words in topics are just there for example and are not a result of a concrete calculation):

- Document 1: 50% “cleaning” and 50% “pet”;
- Document 2: 50% “cleaning” and 50% “pet”;
- Document 3: 50% “cleaning” and 50% “pet”;
- Document 4: 25% “pet” and 75% “alaska”;
- Document 5: 100% “alaska”.

The distribution of words in topics could be:

- Topic “cleaning”: 40% “hoover”, 40% “brush”, 20% collect;
- Topic “pet”: 33% “pet”, 33% “fur”, 16% “animal”, 16% hair;
- Topic “alaska”: 40% “alaska”, 20% “state”, 20% “trade”, 20% “common”.

For existing documents, LDA uses the backtracking approach (assume all documents were previously generated by a model that is using some set of topics and words): each word is randomly assigned to a topic. The algorithm repeats and reassigns the words to topics many times using the Gibbs sampling (distribution generator) until a desired distribution of words per topics is achieved (see Blei et al. (2003) for details). The search query in such a model is also transformed into a document, represented by topic distribution. Using such a query the model will return the documents that have a similar distribution of topics.

### 2.2.3.4 Common Steps in TFLT\textsubscript{IRs}

Review of TFLT\textsubscript{IR} papers suggests that many of them use one or several common steps described below (Marcus et al., 2004; Petrenko and Rajlich, 2013; Scanniello and Marcus, 2011; Zamani et al., 2014):
• **Source code partitioning.** In this step, source code is partitioned into source code entities of selected granularity. Each source code entity is to be represented by textual data and this is gathered during this step.

• **Preprocessing.** During the preprocessing step the data of textual documents, representing source code entities, is filtered and normalized. Usually there are several preprocessing steps:
  
  – Filtering: common words, reserved programming language words (if applied to source code), special symbols, and/or numbers are removed. The common words are usually referred to as “stop words”. For example, “a”, “the”, “is” are stop words. The list of stop words could be different in various implementations. Likewise, the list of reserved programming language words differs per language. For example, in Java, “if”, “for”, and “public” are common reserved words.
  
  – Splitting the words: the identifiers could have lengthy names that consist of several words. For example, it is common to encounter the names such as “sort_list_ascending_order” (4 words) or “isVariableInitialized” (3 words).
  
  – Normalization and stemming: the words are converted in to lower case for convenience and their stems are extracted. For example, the words “sort”, “sorting”, and “sorted” have one common stem “sort”.

• **Applying the IR models** The IR models, described in the previous section are applied to the textual documents and the documents are organized into a search corpus.

• **Querying.** The search query is transformed into a text document and is matched against the search corpus as described in the previous section.

• **Retrieval.** The results (textual documents) are presented in ranked decreasing order according to their relevance to a search query (from the most relevant to the least relevant) and have links to associated source code entities.
2.2 Overview of FLTs

An example of a textual source code TFLT\textsubscript{IR} that aligns with these steps is the work done by Marcus et al. (2004). The authors proposed a TFLT\textsubscript{IR} that splits the source code into source code entities of sub-program level granularity (method level, given the Java context of software systems in their work). The textual data (comments and identifiers) associated with these entities was preprocessed: stop words and special symbols were removed and the words were stemmed. This resultant textual data was associated with the source entities. Finally, the LSI IR model was used to represent and to query the textual corpus of these documents to retrieve results.

2.2.3.5 Hybridization of TFLTs with Other Static and Dynamic FLTs

In this section four examples are given of how TFLTs are mixed with other types of FLTs. These examples include dynamic-textual, structural-textual, dynamic-structural-textual, and textual encompassing both source code and meta-data approaches.

Liu et al. (2007) combined dynamic and textual approaches to FL. In the first step of their approach, a developer creates a scenario that should trigger the execution of a particular feature. After the execution of this scenario, an execution trace is obtained, where each element of the trace is a sub-program call. Prior to scenario execution, source code of a software system is indexed using the LSI approach: every sub-program is represented by comments and identifiers. When the scenario is executed and an execution trace is obtained, a user can use search queries to find interesting sub-programs that are part of this execution trace. The relevant sub-programs are ranked according to their similarity to the user query: the advantage of this approach is reduction of textual search space. The described approach eliminates some disadvantages of both purely dynamic and textual techniques. For example, as was mentioned in Section 2.2.2.2, in dynamic approaches execution traces may contain irrelevant source code entities that have to be eliminated. One approach to eliminate them, is to run several scenarios, that exercise different features, and remove entities that are common to all of these scenarios. An assumption here is that only feature-unique source code entities will remain after several such executions (see Section 2.2.2.2). In the
2.2 Overview of FLTs

approach by Liu et al. (2007), a developer can run a query and obtain a ranked list of relevant sub-programs, eliminating the need for multiple scenario executions. On the other hand, in purely textual FLTs, all source code entities (their textual representations) are compared against a search query. Due to the large numbers of entities, the results are not always precisely accurate. A possible disadvantage though is that: dynamic FLTs are not as good at finding all locations where a feature is executed and so it is possible that the LSI approach is working on a a subset of the code where the feature is implemented. The preliminary analysis of their approach, SITIR, indicated that it is comparable to other state-of-the-art FLTs (Liu et al., 2007).

Scanniello and Marcus (2011) described and implemented a new technique, that combined textual and structural analysis to FL. First, they prepared textual documents using the common steps described in Section 2.2.3.4 and applied VSM for IR, as described in Section 2.2.3.3. Further, they utilized structural information of source code to create a dependency graph of sub-programs (see Section 2.2.2.3). In the graph, each node is a sub-program and the edges represent connections between these sub-programs. Also, in the graph each edge has a weight, where the weight is a cosine similarity between the textual documents, representing sub-programs. When the graph is fully constructed, a BorderFlow algorithm is utilized to partition it into the groups. The clustering algorithm works in a way that allows maximizing connections inside a cluster, while minimizing connections to the outside of the cluster. Finally, each cluster contains a ranked list of sub-programs (represented by textual documents), where similarity between sub-programs is recalculated in the context of the current cluster. The search query is compared against the sub-programs in each cluster and the clusters of sub-programs are returned in ranked order. In other words, if compared to standard VSM, this approach would reorder the returned results: the sub-programs of the most similar cluster come first, followed by the sub-programs of the less similar clusters. The authors conducted an empirical study with five open source software systems using 198 resolved bug reports as a data set. The authors compared this technique to baseline VSM. When combining all the results from all five software systems together, there were statistically significant differences, showing the effectiveness of this approach over baseline VSM.
A FLT may combine dynamic, structural, and textual approaches. For example, in the work by [Dit et al. (2013b)] all these three approaches were employed. The execution scenario was first used to reduce the number of relevant source code entities. Then the textual search, using a query, returned the ranked set of entities, similarly to [Liu et al. (2007)]: only the source code entities in the execution trace were considered. At this stage, web mining algorithms are used to further filter out irrelevant entities. Particularly, HITS and PageRank algorithms were selected for their study (Kleinberg et al., 1999; Page et al., 1999). Both HITS and PageRank algorithms are good at identifying nodes within the graph that have high global importance based on the graph’s structure. On the Internet this means websites that are referred to by many other websites. In software system, these are the entities called by many other source code entities of this system. Usually, such entities with high global importance serve utility/shared needs and are not feature-unique. Therefore, identifying and removing these from the top ranked results might improve the FL. The baseline approach in their study was a hybrid dynamic-textual technique. The empirical study found that the effectiveness of their approach is higher than that of the baseline by 87%.

An example of an approach that employs textual source code and meta-data analysis is that of [Zamani et al. (2014)]. It returns a ranked list of relevant source code entities of file level granularity given a change request. The scores for each of these files is the sum of the two parts:

- The first part is calculated as the sum of the weights of “common nouns”. The common nouns are those that appear in both the summary and/or the description of a new change request and in the identifiers in different versions of the file.

- The second part is also calculated as the sum of the weights of “common nouns”. This time, the common nouns are those that appear in both the summary and/or the description of a new change request and in the summaries and descriptions of older resolved change requests that affected the file.

The weights for common nouns in both cases are calculated as a variation of TF-IDF: the higher scores are given to the common nouns that appear in
2.2 Overview of FLTs

newer versions of a file and in newer change requests. This approach seems to favour the files that change frequently (they are more likely to have corresponding up-to-date change requests), that change to a large extent (many identifiers are created or changed), and that share the similar vocabulary in change requests and in source code. The approach was empirically evaluated using 200 change requests of the four open source projects: JDT, AspectJ, NetBeans, and Rhino. The approach was compared against the baseline VSM. The authors reported that the effectiveness of their approach improved statistically, compared to the baseline VSM.

2.2.3.6 The Advantages and Disadvantages of TFLTs

When compared to dynamic and structural FLTs, TFLTs have some advantages:

- Unlike dynamic FLTs (see Section 2.2.2.2), they can identify features that are always executed.

- TFLTs can identify starting/entry points to a feature and potentially the extent of a feature. In contrast, structural FLTs are dependent on good starting locations (see Section 2.2.2.3). It is difficult using dynamic FLTs to achieve full feature coverage.

- The most simplistic TFLTs, the grep tools, are usually always accessible to developers.

The disadvantages of TFLTs are:

- TFLTs rely on meaningful and abundant lexical information in source code or meta-data. The effectiveness of TFLTs will decrease, for example, if source code is poorly commented, if identifiers are poorly named, or meta-data contains meaningless information.

- The search queries may not match the source code entities (their textual representation) based on different programmers forming the query or, for example, the programmer using a synonym for a term they forget over time.
2.3 How FLTs are Evaluated and The Associated Problems

2.2.4 Trends in Types of Techniques Going Forward

Of the 27 research papers newly found as part of the literature search in this work (see Section 2.2.2.1 Table 2.2), three described exclusively dynamic FLTs, eight described exclusively textual FLTs, and the rest described hybrid FLTs. There seem to be two major trends in the research of FLTs going forward:

- The prevalence of textual source code analysis papers, where this type of analysis is used as either a standalone TFLT or as part of a hybrid FLT.
- The prevalence of hybrid FLTs papers.

Indeed, 53% of papers from [Dit et al., 2013a] and 81% of newly found papers describe FLTs that employ textual analysis. Hybrid FLTs were analysed in 52% of papers identified by [Dit et al., 2013a] and in 59% of newly found papers. Textual meta-data FLTs constitute a small fraction of FLTs. Finally, there is only one FLT that could be described as the “other” type.

The prevalence of TFLTs as standalone or hybrid approaches is likely due to their versatility: their ability to locate ever-executing features (unlike dynamic FLTs), and their ability to locate starting points of a feature (unlike structural FLTs) (see Section 2.2.3.6). However, every type of FLTs has its advantages and disadvantages. To compensate them the FLTs are hybridized ([Dit et al., 2013a] Marcus and Haiduc, 2013). To sum up, the current trend in FLTs is the hybridization of TFLTs with other FLTs.

2.3 How FLTs are Evaluated and The Associated Problems

The output of FLTs is a set of source code entities (for convenience, it is called simply the “result set” in the rest of this work). To assess the effectiveness of a FLT, for every given feature, one needs to compare the result set, with the set of known source code entities for a given feature. This set of known features and known correct mappings to source code entities for these features is called the “gold set” in this work.
2.3 How FLTs are Evaluated and The Associated Problems

Then, as can be seen from Figure 2.3, the set of correct source code entities (called the “correct result sub-set” in this work), identified by a FLT for a given feature, can be obtained by intersecting the gold set and the result set. Knowing these sets, a number of metrics could be applied to evaluate the effectiveness of a FLT. For example, one could calculate the proportion of the correct result sub-set in the result set or the proportion of the correct result sub-set in the gold set.

2.3.1 Identification of the Gold Set

The gold set is usually identified by:

- Experts;
- Non-experts;
- Automated approaches.

Manual identification of the gold set could be performed by both experts and non-experts of a software system (Chen et al., 2001; Ratanotayanon et al., 2010b). Experts are software developers/users experienced in one particular software system. They have enough knowledge about the features and their locations in the source code to form a gold set with a relatively high degree of accuracy. However, such domain experts are not always available. If they are available, they are unlikely to generate a sufficiently scaled gold set required for comprehensive evaluation, because it is a time-consuming task. Non-experts do not have the
2.3 How FLTs are Evaluated and The Associated Problems

equivalent experience in a particular software system. However, their general
knowledge of computer science, and possibly some experience of the system, al-

tows them to locate the features with a certain degree of accuracy. To improve
the accuracy, non-experts usually work in teams of several people. But again, it
is a manual, effort intensive process and a sufficiently scaled gold set is hard to
obtain.

Automated approaches are widely adopted in FL research (Abebe et al.,
\citeyear{abebe2011}; Dit et al., \citeyear{dit2013}; Petrenko and Rajlich, \citeyear{petrenko2013}; Poshyvanyk et al., \citeyear{poshyvanyk2012}).
Such approaches are called the “re-enactment” approaches in this work. The
re-enactment approaches leverage historical change information where textual
descriptions are linked to the code changed. For example, they often rely on ITSs
and VCSs to find links between the change request descriptions of ITSs and the
change-sets of VCSs, where the latter have links to source code entities. The
advantage of the re-enactment approaches is their ability to quickly identify a
potentially correct gold set. However, the change request descriptions are just
an approximation of feature descriptions. Similarly, the source code entities in
change-sets associated with particular change request are just the approximations
of the feature’s extent.

2.3.2 The Problems of Evaluating FLTs

The lack of publicly available gold sets, the lack of details of how these gold
sets are obtained, and of how the result set was preprocessed is a major issue
in the evaluation of FLTs. For example, usually the authors of FLTs describe
the features and the software systems, where these features belong, but do not
explicitly provide the gold set. In those cases when the gold sets are provided,
the authors of FLTs rarely mention the exact versions of software systems (the
exact change-set) for which the gold set was obtained or used. Using the gold
set to find the correct result sub-set, when these two sets come from different
versions of a software system, will most likely yield incorrect results.

Because of the issues mentioned above, the authors of a novel FLT often
cannot compare it with an existing FLT, using the metrics provided in the original
paper of an existing FLT. Instead, usually researchers evaluate FLTs using the following steps:

- Compile a new gold set.
- Select one or few baseline FLTs for comparison. These approaches need to be re-evaluated using the current gold set, because this gold set is most likely different from the one used in the original papers, where these baseline approaches were proposed.
- Assess and compare the effectiveness of the FLT and the novel FLT, based on some set of metrics, using the gold set.

Such an evaluation process has several issues. The gold set has to be re-identified for every new work on FLTs, because of the reasons described above (and if the new authors do not provide the gold set and details such as the exact version of a software system, the cycle continues). Also, the effort of evaluation increases in other ways. For example, instead of focusing solely on the novel FLT they need to evaluate the baseline technique with the new gold set.

Another issue then is that the implementation of the baseline approaches is often unavailable (Dit et al., 2013a). This means that if such an FLT was selected for comparison, it has to be reimplemented by following the instructions in the paper that described this FLT. This could be a subject to misinterpretation leading to the inconsistency in implementation and again requires significant effort.

The solution to this problem seems to be that the FL research community agree on gold sets, the details of obtaining these gold sets (e.g. version of a software system), share the data of the result sets for each new FLT (again detailed) and subsequently make the steps involved in the metrics calculation public and transparent. Another option is to start using the common benchmark suite that would contain the large gold set and recorded results for existing FLTs.

A small step towards such a benchmark was made by Dit et al. (2012). The authors introduced the TraceLab framework that allows for creation of FLTs using the templates and for evaluation of these FLTs using a common gold set.
2.3 How FLTs are Evaluated and The Associated Problems

Unfortunately, this framework is currently very limited: it has the gold set of just five software systems (the gold set was mined using the re-enactment approach) and very limited number of recorded results for existing custom implementations of FLTs. The small size of the gold set could be an issue when assessing the statistical significance of the results. The slow rate of adoption of TraceLab (there are just few works that used it) and potential limitations of the FLT creation templates suggests that this promising initiative needs further evolution and adoption before it realises its full potential.

2.3.3 Metrics Used in the Evaluation of FLTs

When evaluating effectiveness of FLTs, one is often interested in the proportion of the gold set and the result set that are the correct result sub-set (see Section 2.3).

Precision and recall are two metrics that are used to calculate these proportions of relevant elements generated by the approach. Given a gold set \( G \) and a result set \( R \), the precision shows the proportion of correct result sub-set to the result set, whereas the recall shows the proportion of correct result sub-set to the gold set (see Equation 2.5 and Equation 2.6 respectively).

\[
\text{Precision} = \frac{R \cap G}{R} \\
\text{Recall} = \frac{R \cap G}{G}
\]

For example, if a gold set contains five source code entities and a FLT returns 10 source code entities (3 of which are correct), then the recall (measured in percentage for convenience) of such a FLT is 60% and the precision is 30%. Higher recall means that a FLT can locate a higher number of feature related source code entities. The higher precision means that a FLT can suggest source code entities more accurately. Empirical observations of values of these two metrics in many FLTs suggest that these metrics are inversely correlated (Kastner et al., 2014).

These two metrics, however, do not take into account the ranked positions of relevant source code entities and these positions are important when evaluating TFLT IRS: the more correct source code entities in the top positions the better the effectiveness of such FLTs. To address that, two metrics, the effectiveness metric
2.3 How FLTs are Evaluated and The Associated Problems

(to avoid confusion, this metric is called “the effectiveness metric” throughout the text so as to differentiate it from the general term “effectiveness”) and the mean reciprocal rank (MRR) are often used \cite{Poshyvanyk2007}.

The effectiveness metric of a FLT is the position of the first relevant source code entity in the ranked result set \cite{Dit2013,Poshyvanyk2007}, where the result set is ordered in such a way that the most relevant source code entities are supposed to appear on top. Therefore, a lower effectiveness metric’s value would signal a more effective FLT. The rationale behind the effectiveness metric is the assumption that a user will rarely inspect more than one presumably correct entity to obtain a foothold into the feature \cite{Poshyvanyk2007}.

The effectiveness metric is not immediately intuitive, because a higher value signals a less effective FLT. Therefore, it is convenient to employ the MRR metric, which takes an inverse of each of the effectiveness metric’s values over a set of queries $Q$ and returns their mean \cite{Baeza-Yates1999} as shown in Equation \ref{eq:mrr}

$$MRR = \frac{1}{Q} \sum_{i=1}^{Q} \frac{1}{effectiveness_i} \times 100\% \quad \text{(2.7)}$$

The values of MRR will always fall between 0\% and 100\% and higher values will indicate a more effective technique.

The effectiveness metric and MRR consider the position of just the first correct source code entity and disregard the positions of other relevant entities. For example, what happens if a user misses the first correct source code entity? How good then is a FLT at providing other relevant source code entities in the top positions?

To address this, the average precision (AP) metric can be used. First, the sum of precisions (see Equation \ref{eq:ap}) is calculated for all relevant source code entities at every position $k$ in a result set as shown in Equation \ref{eq:ap_sum}.

$$\text{relevant}(k)$$ function returns either one or zero if a source code entity is relevant or not, respectively. For example, in a query that returns 10 ranked source code entities, where the entities in position 1, 2, and 10 are known to be relevant, the AP for this query is equal to $AP = (1/1 + 2/2 + 3/10)/3 = 0.76$. Hence, the larger the
number of correctly predicted answers and the closer they are to the beginning of the ranked result set, the more effective is the technique by this measure.

\[
AP = \frac{\sum_k \text{Precision}(k) \times \text{relevant}(k)}{\text{number of correct entities}}
\]  

(2.8)

Another metric, the mean average precision (MAP) is essentially the AP metric calculated across several queries as shown in Equation 2.9

\[
MAP = \frac{\sum_{q=1}^{Q} AP(q)}{Q}
\]  

(2.9)

2.4 TFLT\textsubscript{IR}s: Problems in the Area

Despite their advantages, TFLT\textsubscript{IR}s have had disappointing returns in terms of the evaluations that have been carried out. These techniques tend to return result sets where each element is in ranked decreasing relevance to a search query (see Section 2.2.3.4). As mentioned in the previous section, in TFLT\textsubscript{IR}s, it is common to report the effectiveness metric of the result set (the positions of first correct source code entities returned). These positions are usually far from the beginning of the result set. For example, in the work by Dit et al. (2013b), judging from the box-plots, the average positions of the first correct entities range from approximately five to approximately 600. In Scanniello and Marcus (2011), they report a mean of 524 for the first correct positions. In yet another example, Bassett and Kraft (2013) report a median of the first correct positions between 108 and 428. More recently TFLT\textsubscript{IR} by Zamani et al. (2014) still report the mean between three and 58 for the first correct positions. These results indicate that a developer has to inspect dozens and sometimes hundreds of source code entities before he or she reaches the first correct one. Also, the lengthy result set evidenced in these studies coupled with the low effectiveness suggests that when a user is interested in the full extent of the feature, a lot of filtering is still required.
2.4 TFLT<sub>IR</sub> S: Problems in the Area

2.4.1 Identifying Best-Performing Configurations in FLT<sub>IR</sub> S

There is a problem of finding the best-performing configurations in FLT<sub>IR</sub> S which also applies to TFLT<sub>IR</sub> S. Typically a FLT is a “one-size-fits-all” approach and authors do not probe the best-performing configurations. In terms of the work that has been done in this area, Panichella et al. (2013) conducted a study to obtain the set of the best parameters for LDA to be used in TFLT<sub>IR</sub> S. More recently, Biggers et al. (2014) studied the influence of LDA parameter on the TFLT<sub>IR</sub>. The authors studied 618 features in six open source Java projects. The major findings convey that the number of topics should increase proportionally to the size of a project, whereas word per topic distribution parameter $\beta$ should proportionally decrease (i.e. inverse proportional to the number of topics). The parameter $\alpha$, which affects topic per document distribution should be set to a constant value of 1.0.

To improve preprocessing of vocabulary, Dit et al. (2011) investigated the impact of applying various identifier splitting techniques to TFLT<sub>IR</sub> S. The authors utilized camel-case, “samurai”, and “oracle” identifier splitting approaches with two TFLTs, where one was solely TFLT<sub>IR</sub> and another a hybrid of TFLT<sub>IR</sub> and dynamic FL. Empirical evidence suggested that the pure IR based technique might benefit when proper identifier splitting is applied but in the case of the hybrid dynamic-textual FLT no significant improvement was observed. Both IR approaches utilized the LSI model. Since comments and identifiers are two main sources of lexical information available in source code, an intelligent identifier splitting might introduce more meaningful keywords into a document corpus. This in turn would allow retrieving more relevant results during the query search process. However, authors reason that when a static textual FL model is used in conjunction with dynamic FL, the latter effectively reduces the search space thus increasing effectiveness of the approach regardless of the identifier splitting being used. This issue is addressed for the FLT proposed in this thesis.

2.4.2 The Vocabulary Paucity Problem in TFLT<sub>IR</sub> S

This problem is core to this research and seems to be common across the IR models (see Section 2.2.3.3). For example, different IR techniques were used with
TFLT_IRs: Problems in the Area

The same poor results: Dit et al. (2013b) used LSI, Bassett and Kraft (2013) used LDA, Zamani et al. (2014) used VSM. Consequently the technique itself does not seem to be the common issue. Instead there seem to be two potential answers as to why this is happening:

- The users (who in these studies are usually developers) are asking the wrong “questions”. In other words, the search query terms are irrelevant to the vocabulary of source code entities.

- There is a lack of meaningful lexicons in source code: the vocabulary paucity problem.

One research direction to address the points above aims at expanding the search queries, suggesting relevant search terms to a developer. For example, the approach by Hill et al. (2009), described in detail in Section 2.2.3.2 and its continuation (Hill et al., 2014b), expand the search query by suggesting the terms that co-occur together in the source code. Abebe et al. (2011) and Abebe et al. (2013) parse the identifiers and create an ontology, where the identifiers are concepts in the ontology: the semantic and structural relationships between these identifiers form the relationships between the concepts in the ontology. These relationships are used to expand the search query. For example, given a search query term, it is matched to an ontology concept (term) and the neighbouring concepts (terms) are proposed for query expansion. Another approach by Rahman and Roy (2015) analyses the change request description and suggests the most relevant terms for a search query. The most relevant terms are those having the highest score in the hierarchy of terms of the change request description according to the PageRank algorithm (see Section 2.2.3.5). Kevic and Fritz (2014) proposed a dictionary approach to translate the change request vocabulary into source code vocabulary. For this task the historical change requests with known links to the source code are analysed and the terms used in source code are mapped to the change request terms.

These query enhancement research directions seem to ignore one problem at play here that might be more fundamental for TFLT_IRs. The IR techniques were designed to retrieve large textual documents (Salton and McGill 1983) and may
2.4 TFLT_{IRS}: Problems in the Area

run into trouble when applied to collections of short texts (Efron et al., 2012). Some of the reasons are:

- Because there is a lesser probability of match for the search query terms in short texts.

- Because the multiplicity of term frequency in TF-IDF (see Section 2.2.3.3) is not leveraged: in short texts, all terms are more likely to appear just once.

These points ring particularly true when considering short source code entities as they seem to be represented by short textual documents. For example, consider the following sub-program (Java method) `has` from the Rhino software system at revision 5fa9e363.

```java
@Override
public boolean has(int index, Scriptable start) {
    if (0 <= index && index < args.length) {
        if (args[index] != NOT_FOUND) {
            return true;
        }
    }
    return super.has(index, start);
}
```

Suppose this source code entity was extracted from the source code and represented by the textual document, as described in Section 2.2.3.4. The document would contain 15 terms (after splitting the identifiers): “has” (2 counts), “index” (5 counts), “start” (2 counts), “args” (2 counts), “script” (1 count), “length” (1 count), “not” (1 count), “found” (1 count). There are no obvious terms to describe this source code entity and these terms are not very meaningful.

Surprisingly, not much work has been done on expanding the lexicons of source code entities. An exception is the approach by Alhindawi et al. (2013) which expands the vocabulary of source code entities with stereotype information. This stereotype information is derived using the static analysis of the role of an entity in a software system. For example, a Java method that returns the value would be annotated using a “get” term.
2.5 How the Change-set Descriptions were Used in FL, BL, IA, and TLR

The problems outlined in the previous section suggest that:

- Augmenting the words associated with source code could be a reasonable thing to do in the expectation that it improves effectiveness.

- Typically, IR techniques present a “one-size-fits-all” evaluation whereby they do not probe different configurations. (The latter can include parameters to IR models, selection of data sources, preprocessing options, and various combinations of such factors.) The hypothesis here is that different configurations may radically alter effectiveness and not always in expected ways. So the configurations should be assessed for the best possible configuration.

These are the core issues addressed in this work and consequently, the following sections predominantly address the first issue but do, in passing address the second issue.

One way in which the vocabulary problem could be ameliorated is the use of change-set descriptions where the words in those descriptions could be linked to the code subsequently changed as identified in a VCS. No-one else has used this VCS data to expand the source code vocabulary in an optimized way for TFLT\textsubscript{IR}, though VCS have been used to some extent in TFLT\textsubscript{s} previously \cite{Chen2001, Cubranic2005, Ratanotayanon2010, Zamani2014}. This approach and the literature that uses change-sets in the context of FL, BL, IA and TLR are now discussed.

2.5.1 Change-sets as a Data Source

This section provides a brief introduction to VCSs. It describes a change-set (defined in Section 1.4), the primary data structure in VCSs, in detail, and also describes change-sets characteristics.
2.5 How the Change-set Descriptions were Used in FL, BL, IA, and TLR

2.5.1.1 The Version Control Systems

There are many project management tools that contain historical/evolutionary information about a software system. The most commonly used are:

- Version control systems (VCS) - these store and track versions of all of the files (not just source code) in a software system.
- Issue tracking systems (ITS) - in the context of software engineering, these systems manage and track the execution of change requests.
- Task management systems - allow the organization of source code pieces around a task.
- Code review systems - allow developers to give feedback on code changes (usually via some links with VCSs).
- Various communication tools - emails, chats, and other tools.

The current research focuses particularly on VCSs. VCSs were initially designed to efficiently manage the evolution of software systems (Rochkind, 1975; Tichy, 1982) and have been identified as an important evolution tool in software change (Buckley et al., 2005). VCSs are third-party tools that are used to store the ordered set of change-sets (defined in Section 1.4) applied to code over time. The data structure that stores the collection of change-sets, along with the file data is often called the “version control repository”. The change-sets in VCSs usually come in chronological order (from the oldest change-set to the most latest) or ancestral order (from parent to child). The version of a change-set is a number or alphanumeric sequence that allows users to uniquely identify the change-set in a collection. VCSs differ in the way the change-sets are organized in a repository.

On the abstract level, some VCSs like Subversion (SVN)\(^1\) could be pictured as a simple singly linked list of change-sets. Other, more sophisticated VCSs, like Git\(^2\) have the structure of a directed acyclic graph as shown in Figure 2.4. Each node in the graph is a change-set created as a result of a commit operation. The

---

1[https://subversion.apache.org/](https://subversion.apache.org/)
2[https://git-scm.com/](https://git-scm.com/)
path from $R_1, R_2, \ldots, R_N$ is called the *trunk* or the *main branch* of a software system. The other branches could be thought of as a parallel development of the software system. The code changes in those branches may or may not end up being part of the software system. For example, the branches $B_{11}$ and $B_{12}$ are used for implementation of code changes and these changes end up being part of a software system: when such a code change is completed, the branch is merged back into the trunk/main branch. Otherwise, the branch could be abandoned and deleted as, for example the $B_{21}, B_{22}$ branch in Figure 2.4.

VCSs differ in the partitioning of their history of changes:

- Change-set per file - every file in a software system has its own history of changes (change-sets) (e.g. CVS\(^1\)).
- Change-set per software system - there is only one line of history of changes for the entire branch of a software system (e.g. SVN).

VCSs also differ in the way the repositories are organized:

- Centralized - there is one central repository (e.g. SVN): where all code changes are stored in one place/repository.
- Decentralized - there are many equal repositories: the code changes are not necessarily shared between these repositories (e.g. Git).

### 2.5.1.2 Change-sets as a Data Structure

A change-set is a basic data structure in VCSs that contains several data fields such as:

- Version - unique identifier of a change-set (see previous section).
- Author - the name of a person who submitted the change-set.
- Time-stamp - the time and date, when the change-set was recorded in a VCS.

\(^1\)http://www.nongnu.org/cvs/
2.5 How the Change-set Descriptions were Used in FL, BL, IA, and TLR

- Description - the textual message that explains the reason for the change-set.

- The list of code changes - the files and places in those files that were modified (how VCSs store and obtain this information is different for each VCS).

This information allows viewers to answer such important questions as: who modified the code, when was the code modified, which part of the code was modified, and why was the code modified.

For instance, in the change-set shown in Figure 2.5, a developer named “Dave Borowitz” modified the code of JGit on Friday 2017-07-28. From the textual description it seems that he has added the in-process fair lock functionality to JGit. After this modification, the new short version (revision) of the master branch is “45da0fc”.

Figure 2.4: Example: a structure of Git VCS.
2.5 How the Change-set Descriptions were Used in FL, BL, IA, and TLR

2.5.1.3 Characteristics of Change-set Descriptions

The focus of this work lies within the change-set descriptions (the line and paragraph starting with “RefDirectory:” in Figure 2.5). These descriptions are textual messages of various length that describe the change: a change that touches source code. In some ways, the source code touched by the change, is described by the description, and this could thus be used to harvest meaningful terms.

The length of change-set description in Git is limited to size_t as defined by the platform. This allows free space as needed, as the default value for size_t is not small. There are no restrictions to the language used in the descriptions allowing for natural language and meaningful content.

There is no guarantee that these attributes imply meaningful feature location content in themselves, but some research has been done on change-set descriptions that impact on their potential ability in this regard: change-set descriptions have their own characteristics with respect to content, quality, classification, and links to other sources:

- *The content and the quality of change-set descriptions.* Though the length of descriptions allows a fairly high number of characters, they seem to be fairly short messages. [Dyer et al., 2013] analysed more than 23,000 software
2.5 How the Change-set Descriptions were Used in FL, BL, IA, and TLR

Figure 2.6: The common types of change-sets as used in different works and the commonality between these types.

Figure 2.7: An example commit of MariaDB.

systems, written in Java, that used SVN and found that 14% of change-set descriptions were empty, 75% of change-set descriptions contained between one and 15 words, and 11% of change-set descriptions were of the size of an “average English sentence” or lengthier. (According to the authors, an average English language sentence consists of 15-20 words.) Though the work
2.5 How the Change-set Descriptions were Used in FL, BL, IA, and TLR

presents several interesting facts about the length of change-set descriptions it does not address the question of how meaningful these descriptions are, particularly with respect to feature location. Moreover, there seems to be no other study that addresses this question of change-set descriptions. Hence, it is an open question as to the utility of these change-set descriptions in feature location.

- **The classification of change-sets.** Not all change-sets that touch the source code files could be useful for the purpose of the vocabulary expansion for source code entities, for example, the change-set given in Figure 2.7. As such it may be convenient to classify the change-sets and to use the groups to determine which change-sets are useful and which are not. Existing work classifies change-sets according to the type of software development and maintenance activities that these change-sets are associated with (Hattori and Lanza, 2008; Hindle et al., 2009). The change-sets are classified using the data of the change-sets’ fields such as description, author, modified files and other (see Section 2.5.1.2). Different authors use different classification but actually refer to similar groupings of change-sets. To avoid redundancy in describing these groups a common classification was used to tie these groups together and describe them serving as a “lingua franca”. This common classification combines the software maintenance types used in the original Swanson (1976) work and also include two other labels: “New feature”, and “Non-programming”. The relationships between the existing classifications and the “lingua franca” classification are shown in Figure 2.6. For the purpose of vocabulary expansion, the only irrelevant group of change-sets is “non-programming”. When applied to source code files this group includes the change-sets that do not affect the logic of source code but instead focus on issues like re-formatting, restyling, code cleanup, modification of licensing, and other similar changes. For simplicity, the latter type of change-sets are called “management” change-sets in the rest of this work. These change-sets have several properties that could be used to help recognize them. Usually they are large change-sets (i.e. touch
2.5 How the Change-set Descriptions were Used in FL, BL, IA, and TLR

Table 2.4: Change requests that could be traced from change-sets.

<table>
<thead>
<tr>
<th>Software System</th>
<th>#Change-sets</th>
<th># Change requests</th>
<th>Change requests traceable from change-sets %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rhino</td>
<td>3312</td>
<td>1082</td>
<td>19.626</td>
</tr>
<tr>
<td>Mylyn.Tasks</td>
<td>8659</td>
<td>8681</td>
<td>73.969</td>
</tr>
<tr>
<td>JGit</td>
<td>3502</td>
<td>945</td>
<td>13.135</td>
</tr>
<tr>
<td>Jetty</td>
<td>10403</td>
<td>3813</td>
<td>35.653</td>
</tr>
<tr>
<td>Ant</td>
<td>13223</td>
<td>5973</td>
<td>17.575</td>
</tr>
<tr>
<td>Hudson</td>
<td>1457</td>
<td>1423</td>
<td>19.767</td>
</tr>
<tr>
<td>JMeter</td>
<td>12135</td>
<td>3292</td>
<td>20.264</td>
</tr>
<tr>
<td>Eclipse.Platform.Text</td>
<td>6273</td>
<td>4355</td>
<td>22.461</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td></td>
<td></td>
<td><strong>27.8</strong></td>
</tr>
</tbody>
</table>

many files) and, importantly, they seem to contain specific vocabulary words (Hattori and Lanza 2008).

- **The links from change-sets to external sources.** Another interesting phenomenon of change-set descriptions is the availability of links to other external data sources. For example, a change-set description may contain the “ids” of ITS change requests or “ID numbers” of code review issues. This potentially allows further expansion past the change-set descriptions allowing annotation of the source code with other relevant data sources. Unfortunately, the extent to which change-sets contain such information seems to be quite small. For example, in the eight software systems that were used later in this work the average number of change-sets that contain references to change requests is 27.8%. This was calculated as the number of change-sets that contain unique change requests’ numbers in their descriptions (see Table 2.4). Hence, though change-sets can link to other data sources, the proportions of such links don’t seem to be high, at least for these eight systems. This is in line with other studies that assessed the number of change-sets containing change request numbers: Brindescu et al. (2014) found that 30.95% of change-sets in 132 VCS repositories (presumably belonging to different software systems) contain change request numbers.
2.5 How the Change-set Descriptions were Used in FL, BL, IA, and TLR

2.5.1.4 Change-set Descriptions in Vocabulary Expansion

Change-set descriptions could form a dataset of lexicons distinct from textual data in source code. The main properties that could potentially make change-set descriptions a useful dataset are their: distinct vocabulary, explicit ties to source code, and multi-language support.

- **Distinct vocabulary.** Developers usually document their code changes as they work on software maintenance tasks ([Maalej and Happel, 2009](http://www.github.com)). Maalej and Happel (2009) claim that this documentation is generally spread across different systems such as ITSs, VCSs, source code, time sheets, and developers’ notes/diaries. Also, they assume that the levels of abstraction used in documentation is different: that comments for source code entity are of lower granularity, more specific, and not as abstract as change-set descriptions or ITSs’ comments. Hence, change-set descriptions could be expected to have a different, higher-level, vocabulary than comments (and possibly identifiers, input-output statements) in source code. Unfortunately though, there seems to be no studies explicitly comparing the vocabularies for the different data sources mentioned above.

- **Explicit ties to source code.** Every change-set links to a set of source code pieces that were modified as part of that change-set. This means that, the links are established between a change-set and every line of code changed as a result of that change-set. This allows for a very exact match between lines of code and a textual description. For example, see Figure 2.8 where every single line of code has an associated change-set to the left. This is different than, for example, comments in source code, which are not explicitly associated with all the lines of code they describe and, indeed, could become detached from the implementation.

- **Multi-language support.** Finally, another property of change-sets that could be beneficial, is their application for FL in heterogeneous systems (i.e. systems written in multiple programming languages). Figure 2.9 (data was taken from GitHub) illustrates that it is common for today’s systems to
2.5 How the Change-set Descriptions were Used in FL, BL, IA, and TLR

be written in many different languages. In contrast, many existing IR based approaches are in fact tailored to work with one particular programming language or languages of the same family (e.g. often times Java and C++) based on their (specific) language parsing requirements. Off-the-shelf they cannot efficiently locate features in multi-language systems due to significant differences in the structure, semantics, and vocabularies of different programming languages (e.g. take HTML and C++). As the same VCS system can be assumed to be employed across the whole system, change-set descriptions can be used to describe all files in the system under version control, regardless of their programming language and structure. However, scoping this for FL in multi-language systems is left for future work and it is provided here solely as another motivation for using the change-set descriptions going forward.
2.5.2 The Usage of Change-set Descriptions in FL, BL, IA, and TLR

According to Table 2.2, there are 11 papers, describing seven unique FLTs, that leverage meta-data as part of their approach. Following the literature search in Phase 2 (see Section 2.1.2.2), another two BL papers (2 unique approaches), five IA papers (3 unique approaches), and three TLR papers (2 unique approaches) were discovered. Also, the technique by Robillard and Dagenais (2008), identified by Rubin and Chechik (2013) is reviewed along with IA approaches. Hence, a total of 15 approaches for FL, BL, IA, or TLR, that leverage change-sets, were discovered.

Probably the two major applications of change-sets in these areas are the study of co-change and the usage of change-sets’ meta-data. The former includes the analysis of source code entities that frequently change together to uncover potential relation between these entities. Since the first description of such an
2.5 How the Change-set Descriptions were Used in FL, BL, IA, and TLR

In the study by Zimmermann et al. (2005), co-changes have been mostly used to support techniques that focus on finding the extent of a feature (e.g., IA) (Ali et al., 2013b; Canfora and Cerulo, 2005; Kagdi et al., 2012), because these techniques rely on evolutionary couplings (Kagdi et al., 2007; van de Laar, 2009). These co-change approaches are excluded from the following review.

Consequently, four FL approaches that use change-set descriptions and two IA approaches that use change-set descriptions are reviewed here. The Table 2.5 summarizes these six relevant approaches remaining. Cubranic et al. (2005), were the first to use meta-data analysis in their “project memory” technique, called Hipikat. The technique would attempt to return all relevant evolutionary documents given a search query. The sources of such documents were VCSs, ITSs, emails, and other available resources. When working on a change request, a programmer had two options when searching for relevant documents: for example, to search using the current source code entity that he/she is working on or to run a search query. In the first case, Hipikat would match the words in the source code entity with any similar documents from the recommender repository and would then return a ranked list of such documents sorted according to their type. In the case of a textual search, a query string is compared against the documents. The authors employ an LSI IR model to score the documents and calculate their relevance. To evaluate the technique, the authors conducted two case studies: one used re-enactment with 20 resolved bugs retrieved from the Eclipse project and the other involved using software developers working on code changes. Though both studies were too small to make any statistically significant conclusions, Hipikat was able to recommend correct locations in many cases. Essentially, the technique enhances program understanding by providing all types of relevant documents. A similar, though more sophisticated approach was proposed by Würsch et al. (2013). In their framework, Hawkshaw, a search query is assisted using natural language terms derived from a software system’s domain ontology. The ontology stores facts about a software system and its evolution (the evolutionary data is extracted from VCS and ITS). Hawkshaw allows users to answer such questions as: “What feature requests where implemented by a developer?” “What are the issues that a developer has commented on?” “How often does a file change?” In fact, these two approaches are not FLTs per se, but...
### Table 2.5: FLT s using change-sets

<table>
<thead>
<tr>
<th>Technique</th>
<th>Research Area</th>
<th>Static</th>
<th>Textual</th>
<th>Meta-data source</th>
<th>IR</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Source code</td>
<td>Meta-data</td>
<td></td>
</tr>
<tr>
<td>Hipikat</td>
<td>FL</td>
<td>+</td>
<td>+</td>
<td>VCS, ITS, e-mail, other</td>
<td>LSI</td>
<td></td>
</tr>
<tr>
<td>Hawkshaw</td>
<td>FL</td>
<td>+</td>
<td>+</td>
<td>VCS</td>
<td></td>
<td>Ontology</td>
</tr>
<tr>
<td>Kayley</td>
<td>FL</td>
<td>+</td>
<td>+</td>
<td>VCS, ITS</td>
<td>VSM</td>
<td></td>
</tr>
<tr>
<td>CVSSearch</td>
<td>FL</td>
<td>+</td>
<td>+</td>
<td>VCS</td>
<td>VSM</td>
<td>PM</td>
</tr>
<tr>
<td>Canfora2006</td>
<td>IA</td>
<td>+</td>
<td></td>
<td>VCS, ITS</td>
<td></td>
<td>Probabilistic</td>
</tr>
<tr>
<td>Zanjani2014</td>
<td>IA</td>
<td>+</td>
<td>+</td>
<td>VCS, ITS,</td>
<td>KNN</td>
<td>Task management system</td>
</tr>
</tbody>
</table>
2.5 How the Change-set Descriptions were Used in FL, BL, IA, and TLR

could assist FL by answering the questions about a software system (Hawkshaw) and suggesting software system’s artefacts (Hipikat). The key difference to the approach proposed in this work is that theirs did not use VCS descriptions to expand the vocabulary of source code entities towards source code identification. Instead they allowed queries to be used to match against VCS descriptions, towards returning specific commits (not code) or they used it to construct a separate ontology that could be queried for specific facts.

The FLT by Ratanotayanon et al. (2010b) (Kayley) combines both structural and “textual meta-data” analysis towards identifying change-sets that are related to a feature description. Specifically, the authors build a search corpus of textual documents, a so-called “transitive change-sets”. The description of a transitive change-set could be a combination of the textual data of a change-set description, the textual data of source code entities, referenced by that change-set (and their structural neighbours), and the textual data of the corresponding change request, if available. Consider, for example, a change-set \(CS\) touching sub-programs \(S_1, S_2, S_3\). Sub-program \(S_1\) calls two other sub-programs \(S_4, S_5\). \(CS\) has a reference to the change request \(CR\). Then the description of the associated transitive change-set \(TCS\) will be a combination of the \(CS\) description, the textual data of \(S_1, S_2, S_3, S_4, S_5\), and the description of \(CR\): in essence, such an approach expands the vocabulary of change-sets. The search query is then matched against the search corpus of transitive change-sets and the output is the ranked list of these transitive change-sets. Several variants of data combinations for transitive change-sets were studied by the authors: change-set descriptions alone, change-set descriptions and textual data of source code entities, change-set descriptions and change requests’ descriptions, and all of the above. Five features from jEdit and Jajuk (the gold set was manually identified by the authors) were selected to evaluate this approach and its data combinations. Kayley, the tool implementing the approach, was implemented as an Eclipse plug-in and was compared with Eclipse’s built-in grep and the FLAT3, the TFLT IR tool by Savage et al. (2010). Though the sample set of features was too small to draw any statistically significant conclusions, still some observations were made:
2.5 How the Change-set Descriptions were Used in FL, BL, IA, and TLR

- Inclusion of textual data of source code entities seemed to introduce noise into the result set and lessened precision (while slightly improving recall).

- The inclusion of change requests’ descriptions seemed to have little effect (and in fact decreased recall in one case).

The first observation is not surprising given the inclusion of the textual data of source code entities that are marginally related to the change-set. For example, it is easy to imagine several utility sub-programs called from a source code entity touched by a change-set. The inclusion of the textual data of these utility sub-programs is unlikely to be relevant to a change-set. The second observation could suggest that the vocabularies of change-sets and change requests are similar to some extent.

But to locate source code entities one has to look for these entities and make them stand out in the search results: not to look for the change-sets and (some-what) associated code. Here lies the major difference between Kayley and the approach proposed in this work: Kayley is “change-set-centric”, the approach in this work is “source code entity centric”. The proposed approach aggregates change-set descriptions around source code entities, giving a more enriched set of vocabulary than Kayley (not just one change-set description, but potentially many associated with the source code entity).

Chen et al. (2001) were the first to explore change-sets of VCSs to support FL. For each line of code, the authors aggregated the descriptions of all change-sets that ever touched that line of code. The textual documents, a result of this aggregation, are stored and queried using VSM. Though this suggests that the technique is aimed at lines of code, the technique and associated tool, CVSSSearch, returned the results at file level granularity, where the score for each file returned is essentially a summation of all scores of lines, retrieved by VSM given a search query, belonging to the file (a higher score is given if relevant lines are closer to each other in a file). Given that relevant change-sets might only change a small (but vital) proportion of the code in a file, it is possible that important input was being diluted and the technique would have been better at lower levels of granularity. The associated empirical study was conducted on five KDE applications by
74 students. Every student was asked to run two queries for each KDE application and the results of these queries were recorded. CVSSearch was “better” than grep if the number of line matches in a relevant file was higher for CVSSearch than for grep and vice versa. The results showed that CVSSearch was “better” in 40% of search queries (statistically significant), compared to 32% in grep, showing a marginal improvement over the most basic of TFLTs. In addition, in terms of feature location, there seems to be a problem with this evaluation: just because the technique has more matches in a file does not make it instantly “better” for feature location. Also, it is not known what kind of queries the students were submitting and what was the purpose of these queries. Though the approach by Chen et al. (2001) is the closest of all existing FLT’s to this work, there are major differences:

- Line of code change-set mapping is inappropriate in terms of granularity: change-sets usually describe many lines of code (27 lines on average, according to Brindescu et al. (2014)), which suggests that a better “real-world” granularity is more appropriate. In this research the annotation happens at the level of FL analysis, creating a more one-to-one mapping between the annotation and the granularity of feature reporting.

- CVSSearch uses change-sets to annotate lines of code, but returns results at the file level, meaning that the input of a specific, potentially relevant, line of code can be overwhelmed by thousands of others that are potentially in the file.

- The CVSSearch aggregates all change-sets en masse without looking on their relevance to the current state of the system or to features in specific. The proposed approach attempts to address this by allowing for aggregation of change-sets in a qualitative, meaningful way: the recentness of change-sets and the types of change-sets.

- The evaluation carried out does not convey its usefulness for FL in specific. Hence, it is hard to tell if change-set descriptions could increase the effectiveness of FL through vocabulary expansion from this work alone.
Canfora and Cerulo (2006) presented an approach to IA that utilizes both change-set descriptions and ITS change request descriptions. When a new change request is submitted to an ITS, the previously fixed change requests textually similar to the one submitted are identified. The textual documents are built for every source code file or line of code that were touched as part of those textually similar change requests. Every textual document is a concatenation of all change-set descriptions and all change request descriptions that touch that file or line of code. The authors used probabilistic IR model and calculated the probabilistic similarity between a change request and the textual documents, representing source code entities (files or lines). Interestingly, the authors do not provide an evaluation of their approach compared to other state-of-the-art approaches: they just routinely report inversely correlated precision and recall. Also they measure the performance of their technique in terms of the time it takes to index the source code. In fact, their technique looks more directed at FL than IA and the result of a search is a ranked list of likely similar files or lines, each of which could be a starting point to a feature.

The approach by Zanjani et al. (2014) builds upon the approach by Canfora and Cerulo (2006), but has several differences. First, the authors extend the approach by adding developer interactions (that come from Mylyn task management system, where these tasks could be associated with change requests via the change request id) and incorporated into the descriptions of source code entities. The authors claim that such an addition improves the IA. However, developer interactions with the code do not necessarily result in code changes: they could be just viewing the code and discounting it. Adding the textual representation of such interactions could include unrelated information and result in increased levels of “noise”. Second, only the source code entities (files and methods) that were changed as part of any previous change requests (recorded in ITS) are included into the search corpus. Hence, many source code entities, will be unnoticed by this approach. Unlike Canfora and Cerulo (2006), the authors used k-nearest-neighbours algorithm (KNN) to find similar code entities. The result of this technique is a ranked list of source code entities. The evaluation was done using four data configurations of the approach: textual data in source code (SIA), change-set descriptions and change requests’ descriptions (ComIA),
all of the above and the textual data of task interactions (InComIA). The evaluation was performed on just one software system and the results showed the statistically significant improvement in the effectiveness when InComIA is used. The approach proposed in this work is different from these techniques, in the following ways:

- Similarly to Chen et al. (2001), Canfora and Cerulo (2006) it maps change-sets predominantly to lines of code. Again this is a questionable mapping as lines of code seem of very small granularity to be mapped to (whole, larger-grained) change-sets. Canfora and Cerulo (2006) do suggest they also can map to files but that is only illustrative, showing that it could be mapped to any granularity.

- Both approaches by Canfora and Cerulo (2006) and by Zanjani et al. (2014) allow for identification of only a fraction of source code entities: those for which an explicit link exists from similar change requests to their associated change-sets (and thus to source code). The approach here allows for identification of all source code entities that have had change-sets applied.

- These techniques combine change-set descriptions with other data such as change requests’ descriptions: it makes it difficult to evaluate the impact of change-sets in isolation. This is particularly acute with respect to Zanjani et al. (2014), where the authors aggregate developer interactions.

- These techniques still aggregate change-sets en masse, whereas the proposed approach allows for the meaningful and qualitative aggregation of change-sets.

- These techniques demand a link from the change request system to the VCS, a link that is frequently not present.

- Also, the evaluation is either non-significant and performance oriented, as in case with Canfora and Cerulo (2006) or uses only one software system, as in case with Zanjani et al. (2014).
2.5 How the Change-set Descriptions were Used in FL, BL, IA, and TLR

Table 2.6: Related work

<table>
<thead>
<tr>
<th>Technique</th>
<th>Approach</th>
<th>Features of CS</th>
<th>Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Data Source</td>
<td>Document retrieval</td>
<td>Context</td>
</tr>
<tr>
<td>Chen et al.</td>
<td>SC;SC + CS + IR(CS)</td>
<td>FL Line</td>
<td>No</td>
</tr>
<tr>
<td>Canfora &amp; Cerulo</td>
<td>CS + CR</td>
<td>IR IA Line/File</td>
<td>No</td>
</tr>
<tr>
<td>Zanjani et al.</td>
<td>SC + CS + CR; SC + CS + CR + I</td>
<td>KNN IA Method/File</td>
<td>No</td>
</tr>
<tr>
<td>Current study</td>
<td>SC; CS; SC + CS</td>
<td>IR FL Method/File</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Legend: SC - source code, CS - change-sets, CR - change requests, I - interactions, KNN - K nearest neighbour algorithm, IA - impact analysis
Table 2.6 summarizes the major differences between this work and those most closely related. Others have done it in a somewhat similar way, but some used an inappropriate granularity, some allow for retrieval of only a sub-set of source code entities, some rely on previously resolved change requests linking to VCS, and some mix questionable data sources, thus making it difficult to tell the value of each individual source contribution.

The technique proposed here differs from the literature reviewed in this section in that it targets source code locations specifically and does so based on analysis at a more appropriate level of granularity. This approach does not rely on a link between the ITS and VCS, a link that the literature reports is infrequent. Finally, it provides coverage over all source code (or at least all source code with change-set data).

Subsequently, the research critically determines the type of change sets that should be employed in the analysis and evaluates the technique in isolation, to assess the true potential of change-set annotation of source code.
3

Aggregating Change-sets for Feature Location

3.1 Overview

As described above, vocabulary paucity is a noted problem in TFLT<sub>IR</sub>s. In the search for a solution to the vocabulary paucity problem, a FLT is proposed, ACIR (Aggregation of Change-set descriptions for Information Retrieval), that leverages change-set descriptions for FL.

There are several different configuration options available to ACIR:

- Whether the technique should be applied at sub-program or file level?

- Whether the technique should include all change-sets when annotating a source code entity or just more recent ones? (Essentially, the question here is: how relevant are the older overwritten change-set descriptions and should they be included?)

- Finally, the type of a change-set may impact the usefulness of information that the description of that change-set contains. For instance, should “management” change-sets (introduced in Section 2.5.1.3) be included in source code annotations.

In this chapter, first the ACIR approach is presented along with a running example, that is given to assist presentation of ACIR. This is followed by a
description of the baseline approach (TFLT\textsubscript{B}), that ACIR will be compared to, and the implementation details for ACIR and TFLT\textsubscript{B}.

## 3.2 The ACIR Approach

A provisional description of the ACIR approach was first presented in (Chochlov et al., 2015). It was later extended and evaluated in (Chochlov et al., 2017), and is presented in detail in Figure 3.1. The ACIR approach allows for aggregation of change-set descriptions to annotate source code entities (using change-sets that have collectively touched a source code entity) for FL and, combining the resultant, textual, source-code descriptions with IR, it allows for indexing, and searching for source code entities, given a search query.

The input to ACIR is source code and a VCS, that stores a set of change-sets $C = \{c_1, \ldots, c_M\}$ on that source code, where $M$ is the number of such change-sets. ACIR relies on the descriptions in $C$ to generate the textual content of documents $D$ for IR, where each document represents a source code entity. There are three major steps in the ACIR approach, as shown in Figure 3.1:

1. Partitioning source code into a set of source code entities $A = \{a_1, \ldots, a_N\}$, where $N$ is the number of entities.

2. Annotating source code entities with change-set descriptions. In this step, ACIR matches all source code entities in $A$ with change-sets in $C$ (that touched those source code entities). ACIR then aggregates textual data from the resultant sub-sets of change-sets (that are mapped to source code entities) to add content to the documents $D = \{d_1, \ldots, d_N \mid \forall d_i, \exists C_i \subset C\}$, where $D$ describes the source code entities in $A$. There are two sub-steps to this step:

   a) Aggregate change-sets for source code entities (that touched those source code entities). ACIR allows for aggregation of change-sets according to the granularity of source code entities, recentness of change-sets in respect to source code entities, and type of change-sets (see Section 2.5.1.3). Depending on the choices made, this results in the
3.2 The ACIR Approach

Figure 3.1: The ACIR approach
formation of different sub-sets of change-sets \( C_i \subset C \) for every source code entity \( a \in A \).

b) Represent source code entities with change-set descriptions (where a sub-set of change-sets was formed for each source code entity, according to selected configuration in the previous step). For each such sub-set, textual descriptions of change-sets are extracted, preprocessed, and concatenated, thus resulting in a set of textual documents \( D \), each element of which representing a source code entity.

3. Use the IR engine to index source code entities (their textual representations: the set of documents \( D \)) and to later search for those source code entities for FL.

### 3.3 Details of Steps in the ACIR Technique

In this section, the steps of the ACIR technique are presented in more detail, describing the transformation of the two inputs to ACIR, the source code and the change-sets, into the textual documents that represent source code entities.

#### 3.3.1 The Running Example

To assist the reader in following the text, an artificial Java method (sub-program in Java language) `public void print(List<Employee>)` is presented as a running example. The method takes a list of employees and prints them. The method undergoes a series of code changes, as shown in Figure 3.2 in descending order, from the oldest to the most recent. The left column in the figure shows the change-sets, where the associated lines of code have most recently been changed. The first change with revision number 59739ca0 has a textual description “Add function to print the employees of a company”. This first version of the method prints all employees from a given list collection. The second revision 40b5b1a9 has a description “Functionality modified to print only those employees that meet certain criteria”. After this modification, the method prints only those employees.
3.3 Details of Steps in the ACIR Technique

Figure 3.2: Example: code changes in print method

that satisfy that certain criteria (embodied in the filter method). The third revision 3ac3d3a6 has a description “All employees are printed in ascending order”. The third revision removes the conditional printing of employees, introduced in the previous code change, and instead prints all employees in ascending order. The last (and most recent) revision 02008e3d has a description “Format code
3.3 Details of Steps in the ACIR Technique

according to new style”. The last modification is an example of a “management” change-set: it does not modify the functionality of the method. It re-formats the code according to the new style.

Table 3.1: The set of change-set descriptions in the running example and their bag-of-words representations.

<table>
<thead>
<tr>
<th>Short id of a change-set</th>
<th>Description</th>
<th>Bag-of-words extracted from a description</th>
</tr>
</thead>
<tbody>
<tr>
<td>59739ca0</td>
<td>Add function to print the employees of a company</td>
<td>&quot;add&quot;, &quot;function&quot;, &quot;print&quot;, &quot;employee&quot;, &quot;company&quot;</td>
</tr>
<tr>
<td>4065b1a9</td>
<td>Functionality modified to print only those employees that meet certain criteria</td>
<td>&quot;function&quot;, &quot;modif&quot;, &quot;print&quot;, &quot;only&quot;, &quot;employee&quot;, &quot;meet&quot;, &quot;certain&quot;, &quot;criteria&quot;</td>
</tr>
<tr>
<td>3ac3d3a6</td>
<td>All employees are printed in ascending order</td>
<td>&quot;employee&quot;, &quot;print&quot;, &quot;ascend&quot;, &quot;order&quot;</td>
</tr>
<tr>
<td>02008e3d</td>
<td>Format code according to new style</td>
<td>&quot;format&quot;, &quot;code&quot;, &quot;accord&quot;, &quot;new&quot;, &quot;style&quot;</td>
</tr>
</tbody>
</table>

The Table 3.1 lists the set of change-sets (in order from the oldest (top row) to the most recent), that are used in this running example, along with descriptions of these change-sets, and the bag-of-words representations that were extracted from the change-set descriptions. The bag-of-words representations were obtained by using a sub-set of the preprocessing steps, described in Section 2.2.3.4. Particularly, for this running example:

- The words in the change-set descriptions were converted into lower case.
- Common stop words were removed.
- The words were normalized, using stemming similar to the Porter stemmer approach (Porter, 1980).

3.3.2 Partitioning of the Source Code

Due to the large scale of modern software, the first step in TFLTs is to partition a software system into a set of chunks of smaller size. Many modern software systems are written in multiple programming languages and contain various configuration, documentation, and other files with meta-data information. The choice of granularity will vary between these different types of files (e.g. configuration
3.3 Details of Steps in the ACIR Technique

files and source code files written in programming languages) and may even be enforced between files of different programming languages. For example, some programming languages, like C, may not have class level granularity, because such constructs are absent in this language.

The granularity of source code entities in FLTs can also be at various levels (see Section 1.1), based on the needs/preferences of the user. This can range from the most fine to the most coarse: line-level (through sub-program level, class-level, and file-level) to package-level.

Some levels of granularity are trivial, in the sense that, the source code of a software system could be easily partitioned into sets of source code entities at that granularity. For example, line level granularity is trivial for partitioning purposes: one simply needs to scan every source code file of a software system line-by-line (possibly eliminating comment and blank lines). File level granularity is trivial as well: one would simply use source code files (as provided by the file system) that are part of a software system. Source code entities of other levels of granularity (e.g. sub-programs) require more sophisticated partitioning. For example, for a sub-program level of granularity, a parser is usually required. A parser transforms the text in source code files into some structuring representation of source code entities (Marcus and Haiduc, 2013) (e.g. an abstract syntax tree is an example of such a representation). ACIR supports the use of a parser to partition the source code into sub-programs, but it also allows for a trivial (file level) partitioning of source code.

Hence, in the ACIR approach, in this first step, the input is the source code. The source code of a software system is then partitioned into a set of source code entities (the output of this step) \( A = \{a_1, \ldots, a_N\} \), where the number of entities \( N \) depends on file or sub-program level granularity, see Figure 3.1.

3.3.3 Annotating Source Code Entities with Change-set Descriptions

This section describes how change-sets are aggregated for source code entities and how the change-set descriptions are then used to textually represent the source code entities.
3.3 Details of Steps in the ACIR Technique

3.3.3.1 ACIR Configurations: Grouping and Classification of Change-Sets

The change-sets’ aggregation sub-step groups change-sets with each source code entity that these change-sets touch. This grouping depends on the selected level of granularity of source code entities. ACIR can also be configured to group change-sets according to their recentness and to ignore “management” type change-sets.

The most recent change-sets, in respect to a source code entity (and in respect to a particular version of a software system), are those that were the last to touch any particular lines of code in that entity. If the most recent change-set is a result of a change request that has other associated change-sets (that touch the same source code entity, but are not the most recent change-sets), then these other change-sets and the most recent change-set are called together “the most recent change-sets by change request”. All historical change-sets in respect to a source code entity are those that affected the entity since it was first introduced/added to a software system.

Finally, ACIR allows for ignoring certain types of change-sets. The types of configurations based on granularity, recentness and type of change-set are presented in Table 3.2. Below, the rationale for selecting these types to focus on is presented:

<table>
<thead>
<tr>
<th>Classification / Grouping</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Granularity</td>
<td>Method</td>
</tr>
<tr>
<td></td>
<td>File</td>
</tr>
<tr>
<td>Recentness</td>
<td>Recent</td>
</tr>
<tr>
<td></td>
<td>All</td>
</tr>
<tr>
<td></td>
<td>/Recent by change request</td>
</tr>
<tr>
<td>Type</td>
<td>Functional</td>
</tr>
<tr>
<td></td>
<td>All (including Management)</td>
</tr>
</tbody>
</table>

- **Granularity values.** One of the main critiques of existing approaches that leverage change-set descriptions (see Section 2.5.2) was the inappropriate usage of granularity for source code annotation. Indeed, annotating a line of code with a change-set description (CVSSearch by Chen et al. (2001))
3.3 Details of Steps in the ACIR Technique

does not seem to be appropriate, because change-sets usually touch many lines of code (27 on average, according to Brindescu et al. (2014)) and their meaning is diluted at such a fine granularity. Hence, mapping source code entities of coarser granularity seems to be a better, more plausible choice, that should be assessed for use with change-set descriptions. But how coarse should these entities be? And how many different granularity types should ACIR support? Given the average number of lines of code touched by a change-set, and the possible spread of these lines of code in several different locations over the code-base, the most intuitive answer would seem to be sub-program. But there are two types of granularity, sub-program and file level, that are predominantly used for FL (Dit et al., 2013a; Marcus and Haiduc, 2013) and this is what the choice is based on.

Selecting granularity coarser than file level (e.g. package level) does not seem to be a reasonable choice: it is very unlikely that a feature is implemented at such an abstract level. Finding an alternative/similar granularity (i.e. coarser than a line of code, but finer than a file) to that of a sub-program is a difficult task: there are not that many granularities of such a level that are common to the majority of programming languages. Hence, it was decided to have a binary choice of values for this configuration: file or sub-program (method).

- **Recentness values.** Another problem with existing approaches, that leverage change-set descriptions, is that they do not consider the recentness of change-sets. In these approaches (Canfora and Cerulo, 2006; Chen et al., 2001; Zanjani et al., 2014), all historical change-sets are aggregated to annotate a source code entity. However, such an approach towards aggregation seems inappropriate. Consider the method `print` in the running example (see Figure 3.2): the change-set `40b5b1a9` adds filtering to the method. The functionality that this change-set has introduced is later removed/modified by newer change-sets. The description of this change-set (if used to annotate the `print` method) is no longer relevant and will likely add noisy textual information (e.g words “certain” and “criteria”). Given how irrelevant the descriptions of such obsolete change-sets could be to the current
state of a source code entity, it seems plausible to assess their relevance for the annotation of those source code entities. Also, a developer may commit changes several times, when working on a change request, every time touching the same source code entity(ies). Thus, it is an open question if it is more effective to use just the most recent change-sets to describe a source code entity or if the change-sets of the most recent change request (one change request can result in several change-sets being committed) should be aggregated. This discussion suggests that three configurations should be assessed: inclusion of all historical change-sets, inclusion of only the most recent change-sets, or inclusion of recent change-sets that are a part of the same change request.

- **Change-set type values.** As it was previously mentioned in Section 2.5.1.3, earlier research of change-sets identified several distinct types of software maintenance, that these change-sets could be related to (Hattori and Lanza, 2008; Hindle et al., 2009). In Section 2.5.1.3, a management change-set was defined as a change-set that is unrelated to functionality modifications in source code. It is rather used to clean the code, format it, update licenses and documentation, reorganize the file structure of a project, reorganize the VCS history and merge branches (Hattori and Lanza, 2008). Such change-sets tend to touch large portion of source code files in a project and thus have the potential to provide lexical descriptions for lots of the source code entities. But, given their lack of association with any specific functionality and their wide scope, it seems reasonable to exclude change-sets of this type from the source code annotation. This assumption is assessed in this work and thus there needs to be a method to remove such change-sets. The choice of values for this configuration is binary: either all change-sets (including management change-sets) are used to annotate a source code entity or only “functional” change-sets (excluding management change-sets) are used.

### 3.3.3.2 Obtaining the Change-sets Configurations

Modern VCSs provide functionality that can support aggregation of change-sets for source code entities to some extent. The aggregation support broadly refers to
any functionality in VCSs that allows for identification and grouping of change-sets that touched a source code entity at any time during a software system’s evolution. The extent of this support, however, varies between VCSs and it is difficult to account for change-set aggregation peculiarities of every VCS. (It is even more difficult to account for various tools that can extend the functionality of such a VCS by adding aggregation support). Table 3.3 summarizes the aggregation support that is important for ACIR configurations in three VCSs: Git, SVN, and CVS. These three VCSs were picked because they frequently appear in research literature related to FL, BL, IA, and TLR and there are no reliable statistics regarding the popularity and usage distributions of VCSs.

As can be seen from the table all of these VCSs allow for aggregation of all historical change-sets and the most recent change-sets at file level granularity. Though SVN and CVS do not support the aggregation of the most recent change-sets at sub-program level, the problem could be solved in the following way: one needs to aggregate the most recent change-sets at file level and then select the range of lines of code of those sub-programs. However, only Git supports aggregation of all historical change-sets at sub-program granularity level. For SVN and CVS this cannot be trivially solved using existing aggregation support in these VCSs: a sub-program in the current revision will need to be compared with the same sub-program in the previous/next revision, and iteratively in similar manner in all revisions going backward/forward to identify if a particular revision has modified that sub-program (also the positions of sub-programs need to be tracked across revisions). Finally, none of these three VCSs (and none of the existing VCSs, known to this author) support the aggregation of change-sets according to their type and the aggregation of change-sets by a common change request (hence, this aggregation support is completely omitted in Table 3.3).

Therefore, the following reasoning could be used for the aggregation support:

- For granularity and recentness: if a VCS supports all aggregation combinations (e.g. Git), then it is reasonable to rely on that VCS, in regards to this support. Otherwise, a generic aggregation approach, as described below in Section 3.3.3.3, is required.

\[1\] The implementation of ACIR, used in this work, relies on Git
3.3 Details of Steps in the ACIR Technique

- For aggregation by change request and type, a manual solution, as described in Section 3.3.3.4 and in Section 3.3.3.5 respectively, is required.

Table 3.3: Change-sets’ aggregation support in VCSs: Git, SVN, and CVS

<table>
<thead>
<tr>
<th>Source code entity (granularity)</th>
<th>Aggregation</th>
<th>VCS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Git</td>
</tr>
<tr>
<td>File</td>
<td>All historical</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>Recent</td>
<td>Yes</td>
</tr>
<tr>
<td>Sub-program</td>
<td>All historical</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>Recent</td>
<td>Yes</td>
</tr>
</tbody>
</table>

3.3.3.3 Aggregating Change-sets by Granularity and Most-recent/All Historical Change-sets

Existing research tried to address the problem of aggregation of all historical change-sets at a finer level of granularity than file level, but did so for line-of-code granularity only (Canfora and Cerulo, 2006; Chen et al., 2001; Zimmermann et al., 2006). The existing approaches are very much alike and propose to:

- Use the “diff” tool\(^1\) to calculate changes in a line of code (position in a file and content).

- Apply diff repeatedly to a set of change-sets, going backward (Chen et al., 2001), starting from the most recent change-set, or going forward (Zimmermann et al., 2006), starting from the oldest change-set, to construct a data structure, that tracks the history of modifications in lines of code across change-sets.

In VCSs, given two change-sets, the diff tool allows for identification of lines of code in all source code files (that are tracked by a VCS) that were:

- Inserted;

\(^1\)In this work, the diff tool refers to any tool, that allows for calculation of differences between two data sources (such as files or change-sets).
3.3 Details of Steps in the ACIR Technique

- Deleted;
- Modified.

It should be mentioned that some very basic variations of diff do not recognize modifications of a line of code. For example, Figure 3.3 shows the usage of Git’s diff to compare the most recent change-set 02008e3d, from the running example, with its predecessor, the change-set 3ac3d3a6 (the “-” signs demark the lines that were deleted, and the “+” signs demark the lines that were added). In this example, the first line of the diff’s output (containing the signature of the print method) is shown as deleted and it is then shown again as added (as seen in the second line of the output), although it was only slightly modified (and thus should probably be marked as modified). Because tracing modifications is important for change-set aggregation (otherwise, how would one trace the evolution of line(s) of code beyond the most recent evolution), both Chen et al. (2001) and Canfora and Cerulo (2006) propose to trace these line modifications using the following heuristics:

![Figure 3.3: Example: applying diff to compare the change-set (02008e3d with the change-set (3ac3d3a6).](image)

- Inspect only those additions and deletions, where positions of lines overlap (in the old version and in the new version).
- Match highly similar lines of code inside those overlaps and mark them as modified (for example, Levenshtein distance could be used to measure similarity between two lines of code).
Applying diff repeatedly to a set of change-sets will generate a history of line modifications (in a file). This history could be stored in a collection of arrays (where each array stores each lines of code as per a change-set) (Canfora and Cerulo, 2006; Chen et al., 2001) or in a multipartite graph (Zimmermann et al., 2006). For example, applying the approach by Zimmermann et al. (2006) to the method in the running example would generate the line history as shown in Figure 3.4. In the figure, the circles are the lines of code. The letter “M” inside a circle means that a line was modified, the letter “A” means that a line was added, an empty circle means that a line remained unchanged, and the horizontal edges between the circles allow viewers to trace the position of a line (if a line has no edges going to the next change-set it means that a line was deleted in the next change-set).

![Figure 3.4: History of changes for lines of code in print method, from the running example.](image)

The approach by Zimmermann et al. (2006) can be modified towards a generic aggregation model, that satisfies all configurations of ACIR, as shown in Fig-
In this model, every file, taken at some particular version, is represented by a list of $N$ lines of code. Every line of code (every member of the list) points to a linked list of arbitrary size that captures the evolution of this line. Thus, every node in the linked list is a reference to a change-set that modified this line. Each change-set reference points to a hash table of $M$ change-sets (the sum of the arbitrarily-sized linked lists, over the $N$ lines of code) to quickly lookup the entry. Such a model allows for efficient storage of changes for each line of code (only modifications and insertions for line of code are considered) and allows for versatile change-set aggregation configurations (including aggregation by a change request and type).

The examples below illustrate how this model could be used to aggregate change-sets for the various configurations, as described in Table 3.3:

- **Example 1**: Find the most recent change-sets of a file. If there is a file consisting of $N$ lines of code, then for each line of code the first node
of the associated linked list is taken. The references from these nodes are collected and aggregated to produce the resulting set of change-sets.

- **Example 2: Find the most recent change-sets of a sub-program.** Suppose a sub-program starts at line $K$ and ends at line $L$ in a file that has $N$ lines of code, so that $1 \leq K \leq L \leq N$. Then for each line of code between $K$ and $L$ inclusively, the first node of the associated linked list is taken.

- **Example 3: Find all historical change-sets of a file.** For each line of code of $N$ lines, all the nodes of the associated linked list are taken. The references from these nodes are collected and aggregated to produce the resulting set of change-sets.

- **Example 4: Find all historical change-sets of a sub-program.** Suppose a sub-program starts at line $K$ and ends at line $L$ in a file that has $N$ lines of code, so that $1 \leq K \leq L \leq N$. Then for each line of code between $K$ and $L$ inclusively, all the nodes of the associated linked list are taken. The references from these nodes are collected and aggregated to produce the resulting set of change-sets.

### 3.3.3.4 Aggregating Change-sets of a Change Request

When working on a change request a developer may commit several times. Hence, considering only the most recent change-sets that touched lines of code may be inappropriate/incomplete in terms of that change request and, it may only address one aspect of that change request. If the change request is something like a feature enhancement, this means that the change-set will be incomplete in terms of the feature’s code, and quite possibly in terms of the feature description. Hence, this would be an interesting configuration to study. The approach then would be to combine the change-sets of the most recent change request that affected a source code entity and use these to describe this entity. This boils down to the question: if there are $N$ change-sets $C_0 \ldots C_{N-1}$ that touched a line of code, ordered from the most to least recent ($C_0$ being the most recent change-set), how would it be
3.3 Details of Steps in the ACIR Technique

Figure 3.6: Aggregating by branches

It is possible to find any other change-sets (if there are any) that together with $C_0$ could belong to the same change request?

One heuristic is to use the structural features of VCSs. When working on a change request, it is a common practice to create a development branch and commit all the changes relevant to a change request to that branch. When the change request is completed, this branch is merged back into the main branch (see Figure 3.6). Hence, all the change-sets that are part of a development branch, but not a part of the main branch might be considered as belonging to a change request. In some VCSs, like SVN, the process of identification is straightforward. Every branch in SVN is just a copy of a main branch, resting in its own folder. Hence, the branch specific (change request specific) change-sets are those not encountered in the main branch. In other VCSs, like Git, identification of branches is more complicated. Version history in Git has the form of a directed acyclic graph, where each node could have between one or two parents (0 parents in a special case of an initial change-set).

In such instances, a depth-first graph traversal algorithm could be modified and employed to identify branches. Particularly, one would walk all possible paths, starting from the most recent node in a graph (traversing backwards) and marking nodes as visited along the way, saving points of path split (nodes that have two parents), until the initial change-set (0 parents) or an already visited change-set is reached. When an initial/visited change-set is reached the algorithm would backtrack to the saved split node and start traversal from there. When all paths have been walked, it is time to decide which are the development branches and which is the main branch. This could be done by inspecting the change-sets. For example, if they all have the same author and short inter-commit periods
3.3 Details of Steps in the ACIR Technique

then it is more likely that it is a development branch.

For example, in Figure 3.6, there are two alternatives:

- 8, 7, 5, 3, 2, 1 is the main branch and 6, 4 is the development branch;
- 8, 7, 6, 4, 2, 1 is main and 5, 3 is development branch.

These combinations need to be checked to find out which is the development branch. Going back to the initial question: it could be checked if \( C_0 \) belongs to either alternative branch and, if it does, check all other remaining \( N - 1 \) change-sets that touch that line of code for their membership in that same branch and thus group them with \( C_0 \).

However, branches are not always available: empirically, it was noticed (the author noticed this, when manually scanning the commit-histories of several software projects) that in open source projects it is very common for a single person to work on a particular small change request and commit changes within certain, often short, periods of time directly to the “main” development branch. Hence, clusters with similar author and committed within short periods of time could potentially identify change-sets relevant to a change request. For example, change-sets C4 and C3 in Figure 3.7 are likely to belong to the same change request, because they were committed by the same author A within five days. C3 and C2 are unlikely to belong to the same change request: despite having the same author A, as they were committed four months apart of each other. C2 and C1 were committed just three days apart, but have different authors, and hence are less likely to belong to the same change request. So a second heuristic could involve iterating through \( N \) change-sets, starting from \( C_0 \) and grouping those change-sets meeting these author/time-scale criteria.

Figure 3.7: Aggregating by author and period between change-sets
Since neither approach guarantees exhaustive identification of change-sets belonging to a change request, they could be combined to improve the accuracy. First, the branching approach is used and if there are no branches available, then author/time-stamp analysis steps in.

Finally, no reliable literature was found on how often developers commit when working on a specific change request. Direct observations of change-set history suggest that in many cases these are short periods between one and eight days. As a result of the observations of change history, a 10 day upper limit was selected for associating change-sets to a change request in this study, but it is acknowledged that this is a somewhat arbitrary choice which should be investigated.

### 3.3.3.5 Aggregating Change-Sets by Type

As mentioned in Section 3.3.3.1, the inclusion of "management" change-sets during the aggregation step should be assessed. Hattori and Lanza (2008) noticed that change-sets of this type usually share a set of common keywords. This finding is leveraged to implement the filtering of management change-sets: a naive text classification approach could thus rely on such keywords to spot management change-sets. The following filtering keywords, suggested by Hattori and Lanza (2008), are used for this purpose in this research: \{"clean", "license", "merge", "release", "structure", "integrate", "copyright", "document", "manual", "javadoc", "comment", "migrate", "repository", "code review", "polish", "upgrade", "style", "format", "organ", "todo"\}. Change-sets with any of these keywords in their descriptions are marked as management change-sets.

### 3.3.3.6 Representing Source Code Entities with Change-set Descriptions

The aggregation sub-step returns a set of change-sets for a source code entity. However, many of these change-sets are duplicates, because it is usual for a change-set to describe several lines of code in the same entity (sub-program or file). Therefore, as a precursor to textual representation of source code entities, the duplicates need to be removed using a distinct function \text{Distinct}(C_i).
3.3 Details of Steps in the ACIR Technique

For every \( c \in C_i \) the terms of change-set descriptions are extracted and added to a set of terms \( T_i \). The content of every \( d_i \in D \) is composed of the corresponding \( T_i \). Repeating terms in \( T_i \) are not removed therefore every \( d_i \in D \) is represented by a bag-of-words. The textual data for each document is then pre-processed:

1. Special characters and numbers are removed.
2. Words are converted into lower case.
3. The stop words are removed (see Section 2.2.3.4). (Change-set descriptions are written in natural language and therefore do not require removal of programming language specific keywords.)
4. Change-set descriptions could contain the names of source code entities. These names should be split if required. For example, there are many multi-word Java source code names, that are written using camel case or dot notation. Alternatively, in the Python language, for example, it is common for these names to contain underscore notations.
5. Porter stem filtering is applied to all resultant terms (Porter, 1980).
6. Short words of less than three characters are removed.

3.3.3.7 Example: Annotating the Method in the Running Example Using Various Configurations

Table 3.4 demonstrates how different annotation configurations for \textit{print} method in the running example, result in different sets of change-sets and subsequently different textual representations. The granularity in this example is set to method (sub-program) level. Configuration values in the first column are used as presented earlier in Table 3.2. For the sake of this example, it is assumed that change-sets \textit{3ac3d3a6} and \textit{40b5b1a9} belong to the same change request.
### 3.3 Details of Steps in the ACIR Technique

#### Table 3.4: Example: Annotating the Method in the Running Example Using Various Configurations

<table>
<thead>
<tr>
<th>Aggregation configuration</th>
<th>Set of distinct aggregated change-sets ((C_i))</th>
<th>Resultant textual document ((d_i)), represented as bag-of-words</th>
</tr>
</thead>
<tbody>
<tr>
<td>Recent/Functional</td>
<td>3ac3d3a6, 59739ca0</td>
<td>&quot;employe&quot;, &quot;print&quot;, &quot;ascend&quot;, &quot;order&quot;, &quot;add&quot;, &quot;function&quot;, &quot;print&quot;, &quot;employe&quot;, &quot;company&quot;</td>
</tr>
<tr>
<td>Recent/All</td>
<td>02008e3d, 3ac3d3a6, 59739ca0</td>
<td>&quot;format&quot;, &quot;code&quot;, &quot;accord&quot;, &quot;new&quot;, &quot;style&quot;, &quot;employe&quot;, &quot;print&quot;, &quot;ascend&quot;, &quot;order&quot;, &quot;add&quot;, &quot;function&quot;, &quot;print&quot;, &quot;employe&quot;, &quot;company&quot;</td>
</tr>
<tr>
<td>Recent by a change request</td>
<td>3ac3d3a6, 40b5b1a9, 59739ca0</td>
<td>&quot;employe&quot;, &quot;print&quot;, &quot;ascend&quot;, &quot;order&quot;, &quot;function&quot;, &quot;modif&quot;, &quot;print&quot;, &quot;only&quot;, &quot;employe&quot;, &quot;meet&quot;, &quot;criteria&quot;, &quot;add&quot;, &quot;criteria&quot;, &quot;employe&quot;, &quot;company&quot;</td>
</tr>
<tr>
<td>All historical/Functional</td>
<td>02008e3d, 3ac3d3a6, 40b5b1a9, 59739ca0</td>
<td>&quot;format&quot;, &quot;code&quot;, &quot;accord&quot;, &quot;new&quot;, &quot;style&quot;, &quot;employe&quot;, &quot;print&quot;, &quot;ascend&quot;, &quot;order&quot;, &quot;function&quot;, &quot;modif&quot;, &quot;print&quot;, &quot;only&quot;, &quot;employe&quot;, &quot;meet&quot;, &quot;criteria&quot;, &quot;add&quot;, &quot;function&quot;, &quot;print&quot;, &quot;employe&quot;, &quot;company&quot;</td>
</tr>
<tr>
<td>All historical/All</td>
<td>3ac3d3a6, 40b5b1a9, 59739ca0</td>
<td>&quot;employe&quot;, &quot;print&quot;, &quot;ascend&quot;, &quot;order&quot;, &quot;function&quot;, &quot;modif&quot;, &quot;print&quot;, &quot;only&quot;, &quot;employe&quot;, &quot;meet&quot;, &quot;criteria&quot;, &quot;add&quot;, &quot;function&quot;, &quot;print&quot;, &quot;employe&quot;, &quot;company&quot;</td>
</tr>
</tbody>
</table>

#### 3.3.4 Indexing Documents

In the last step, the corpus of documents \(D\) has to be stored and queried. ACIR is using VSM to index and query these textual documents. VSM was selected, because it is well-studied and effective (Lee et al., 1997).

Other IR models (such as LSI and LDA) have been applied to assist in FL and in related fields (Biggers et al., 2014; Lukins et al., 2008; Panichella et al., 2013), but also were repeatedly reported by different authors to be less effective than VSM (De Lucia et al., 2012; 2014; Rao and Kak, 2011). The authors explain this by the low diversity of concepts in the documents, extracted from source code.
entities. Hence, the advantage of LSI and LDA is yet to be proven in this context, and it seems that the VSM approach is still a more preferable option. Thus, in this work VSM is used along with a TF-IDF function to calculate document statistics for IR, and cosine similarity to compare documents.

3.4 The Baseline Approach: TFLT\textsubscript{B}

Several existing approaches were considered for TFLT\textsubscript{B}. However, all of them had to be abandoned for the following reasons:

- One option for TFLT\textsubscript{B} was to use grep, similar to Chen et al. (2001). However, this approach is inappropriate, because pattern matching and VSM are very different, rendering results produced by these two approaches incompatible.

- The approach by Zanjani et al. (2014) (along with their baseline approach) was not publicly available.

- The FLAT3 approach by Savage et al. (2010) had inappropriate granularity: a mix of a sub-program and a line of code (field) granularity and no file level granularity.

- TraceLab still has a low adoption rate (only two papers of 27 found between 2011 and 2015 report on using this framework).

Hence, it was decided to use the steps to TFLT\textsubscript{IR} approach, described by Marcus et al. (2004), but also to modify these steps according to the recent best practices in TFLT\textsubscript{IR}s. Particularly, the following modifications were introduced:
3.4 The Baseline Approach: TFLT$_B$

- VSM was used to align with ACIR (the original approach by [Marcus et al. (2004)] used LSI).

- Identifiers, comments, and literals were used to describe source code entities. It was reported by [Biggers (2012)] that such combination of textual data improves effectiveness. This combination was also used by [Alhindawi et al. (2013)] in their work on vocabulary expansion using stereotypes. In contrast, [Marcus et al. (2004)] used only identifiers and comments.

- Text preprocessing was improved. [Dit et al. (2011)] suggest that better splitting techniques could improve the effectiveness of a TFLT. Hence, the TFLT$_B$ also supports “dot” splitting, along with the “camel case” and “underscore” splitting originally used by [Marcus et al. (2004)]. Also, when file level of granularity is used, the license information is removed from the content.

TFLT$_B$ follows these steps below to construct the search corpus for FL (see Figure 3.8):

1. **Partition.** Source code is first partitioned into a set of source code entities of file or sub-program level granularity.

2. **Generate documents’ content.** The content of every document, describing a source code entity, is comprised of the set of terms, which are derived from the identifiers, comments, and literals contained in the source code for that entity. The text preprocessing is similar to that described for ACIR (see Section 3.3.3.6), but additionally, programming language keywords were removed. After this step, all the documents in the search corpus are represented by bags-of-words.

3. **Index documents.** The corpus of documents is then stored using the VSM IR model.

---

1 The author observed that licensing information is common in source code files and introduces noise to the textual data.
3.5 Implementation details

Due to the nature of operations involved, the implementation of ACIR and TFLT$_B$ relies on several third party libraries. Particularly, library support is needed to communicate with a VCS repository, to parse the source code of a subject system, and to index and search the resulting documents using the IR engine. The separation of concerns in the design of both ACIR and TFLT$_B$ potentially allows them to be applied to a wide range of software systems irrespective of programming language and VCSs, used by those systems. However in this work, the implementation of ACIR and TFLT$_B$ is restricted to be used with systems implemented in the Java programming language only and using Git VCS. The IR part uses VSM and TF-IDF similarity [Salton and McGill 1983]. Below a list of the essential operations and third party libraries, that support them, are provided:

- **Java code parsing.** To support Java code parsing, `JavaParser` was used. This tool is employed to split Java source code into the source code entities of selected granularity. Also, it allows for extraction of the textual content from such entities. When partitioning a system at method level of granularity, abstract methods, constructors, and methods of anonymous classes are excluded from the set of source code entities. Abstract methods are excluded, because they do not contain functionality. Constructors are excluded, because they set the initial state of the object, but do not implement functionality per se. Anonymous methods are excluded to avoid redundancy: they are nested inside other methods. At the file level of granularity unique entities are identified using their path names, whereas at the method level of granularity, return type, method name, class name, and parameter types are used for unique identification.

- **Interaction with Git VCS.** To interact with Git, the `JGit` library was used. The functionality this library allows for retrieval of change-sets from Git repositories, the establishment of links between change-sets and source code, 

1. [https://github.com/javaparser/javaparser](https://github.com/javaparser/javaparser)
2. [https://eclipse.org/jgit/](https://eclipse.org/jgit/)
3.5 Implementation details

and the identification of code changes between two different snapshots of code in Git. To retrieve all historical change-sets at method level granularity, `git log -L` command was employed, since such functionality is not yet available in JGit.

- **IR.** To support IR operations, *Lucene*[^1] a text search library, was employed, that supports the VSM IR model for the indexing and retrieval of text. Lucene introduces certain modifications to the TF-IDF statistics and similarity formulas as described in their official documentation[^2].

Also, to support the research method, a tool was required to interact with *Bugzilla*[^3] ITS repository to retrieve change requests. For this task, the *B4J*[^4] tool was used, which allows the retrieval and analysis of change requests.

The TFLT_B, ACIR, and support tools were implemented as the *FUSIX* library (FUsed code Search based on Information retrieval of Change-sets and Source code data → FUSICS → FUSIX), written in Java, which is available on-line[^5]. A source code example of using the FUSIX API is included in the appendix, see [Appendix B: Source Code Example of Using FUSIX API](#). The screen-shot of the demo application is also included in the appendix, see [Appendix C: Screen-shot of Demo Application](#). The application itself with data from the empirical studies in this thesis, is accessible at this address[^6].

[^1]: <https://lucene.apache.org/core/>
[^2]: <https://lucene.apache.org/core/4_2_0/core/org/apache/lucene/search/similarities/TFIDFSimilarity.html>
[^3]: <https://www.bugzilla.org/>
[^4]: <https://techblog.ralph-schuster.eu/b4j-bugzilla-for-java/>
[^5]: http://www.fusix.co
[^6]: http://demo.fusix.co
4

Experiment Procedures

4.1 Overview

In the previous chapters, it was argued that change-sets could potentially improve FL by expanding the vocabulary of source code entities. However, they have not been thoroughly studied for this purpose in isolation and nor has the best-performing configuration of change-set aggregation been determined. In Chapter 3, ACIR was proposed: a technique that allows for the use of change-set descriptions to annotate source code entities for FL. ACIR allows for aggregation of change-sets (for annotation) in various combinations. The objectives of the empirical evaluation presented in this work is finding the best-performing configuration of ACIR and evaluating its performance when compared to a baseline state-of-the-art approach TFLT_B.

In this chapter an empirical design towards this objective is presented. First, the design of a pilot study is reported upon. Pilot studies have been suggested as an important first step to a good empirical study (Van Teijlingen and Hundley 2002). Pilot studies allow for:

- Assessing the feasibility and adequacy of the proposed research tools.
- Anticipating the risks of the main study.
- Consequently refining the design of a study.
4.2 Design of the Pilot Study and How it Shaped the Scaled-up Study

Usually pilot studies are merely descriptive and do not convey statistically significant results, which is also the case in this work. Consequently, it was used to refine ACIR, the empirical design, and to get a feel for the effectiveness of ACIR approach only.

Further in the chapter, an experiment design used in the final evaluative study is described. The subject systems and the strategy by which they were selected are introduced. These are the software systems to which ACIR and TFLT_B will be applied to locate features. After the results of applying these approaches for FL are obtained, a gold set (see Section 2.3) is needed to assess the effectiveness of these results. In this work, an automated re-enactment approach is used to construct the gold set, and this too is described. Further, the metrics that are used to evaluate the results are presented and, finally, the replication procedures are described.

4.2 Design of the Pilot Study and How it Shaped the Scaled-up Study

4.2.1 Methodology of the Pilot Study

The methodology of a pilot study is a predecessor to the methodology of a scaled-up study discussed later in this chapter. Thus, it shares some aspects of that methodology such as experiment work-flow (see Section 4.3), a sub-set of the subject systems used (see Section 4.4), the re-enactment approach for building a gold set (see Section 4.5), and the metrics employed. The methodology is different in the following aspects:

- A sub-set of the systems used in the scaled-up study, was used: Rhino and Mylyn.Tasks were selected as subject systems. These systems were selected, because they were previously studied by other researchers when evaluating FLTs [Kevic and Fritz 2014, Zamani et al. 2014] and several size and activity characteristics for these systems are presented in more detail in Table 4.1. As seen from the table, for each level of granularity, two distinct cases were analysed: the most recent or all change-sets. The average number
4.2 Design of the Pilot Study and How it Shaped the Scaled-up Study

Table 4.1: Subject systems of the pilot study

<table>
<thead>
<tr>
<th>Subject System</th>
<th>Initial release</th>
<th># Files</th>
<th># Methods</th>
<th>Average methods per file</th>
<th>LOC</th>
<th># Commits (master)</th>
<th>Average of distinct commits per artifact</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rhino (rev. 06710fa1)</td>
<td>Autumn 1997</td>
<td>468</td>
<td>6120</td>
<td>13.07</td>
<td>90435</td>
<td>3136</td>
<td>2.11 3.56 9.44 15.44</td>
</tr>
<tr>
<td>Mylyn.Tasks (rev. 68400a6f3)</td>
<td>June 2005</td>
<td>1086</td>
<td>8517</td>
<td>7.84</td>
<td>117348</td>
<td>8576</td>
<td>2.27 3.34 11.48 15.54</td>
</tr>
</tbody>
</table>

of distinct commits per source code entity are comparable across systems for each granularity level, except when file level granularity (including the most recent commits only) is considered: 9.44 for Rhino, compared to 11.48 for Mylyn.Tasks. Unsurprisingly, the method level granularity with the most recent change-sets shows the lowest average number of distinct change-sets for the observed projects, while the file level granularity where all change-sets are retrieved shows the highest average number of distinct change-sets.

- With respect to the pilot study, ACIR was inexactly\(^1\) compared to existing state-of-the-art TFLTs as a preliminary assessment of its utility. These techniques use different metrics: most commonly MAP, the effectiveness metric, and MRR. Hence, in addition to the MAP metric, the effectiveness metric and the MRR metric were used in the pilot study to evaluate ACIR, and allow it be compared against a wider pool of techniques.

- Because the pilot study’s goal is to provisionally assess the technique, smaller gold sets were obtained: 19 for Rhino and 20 for Mylyn.Tasks.

4.2.2 Threats to Validity and Lessons Learned

This section presents the threats to validity that are pilot-study specific. For the list of common threats to validity, that apply to across experiments in this work, see Section 7.3. The threats to validity are organized into construct, internal, external, and conclusion (where applicable) as per Wohlin et al. (2000).

\(^1\)Different subject systems or their versions were used for comparison.
There are several issues that could have affected the validity of the pilot study. The primary threat is one of construct validity. The study involves indirect comparison of ACIR to other TFLT_{IR}s in the literature. This limitation was addressed in the scaled-up experiment by implementing a standard textual, IR-based FLT (TFLT_{B}) for comparison.

Also, another construct validity issue is that the vocabulary of source code entities might get polluted, where one change-set is associated with many bugs or where a change-set is associated with large management task affecting many source code entities, but possibly not directed at features specifically. An example might be updating the licences in the source code. The issue led to additional configuration refinements to ACIR being considered in the scaled-up study: aggregation of change-sets by a change request and filtering of management change-sets were added.

A threat to external validity can come from the low statistical power of this study: in this observational study two subject systems and 39 ITS change requests were analysed. This is an exceedingly small cohort of systems and features, and the scaled-up study should address these issues.

To sum up, the lessons learned in the pilot study allowed for:

- Refinement of ACIR approach: aggregation of change-sets by change request was added as one of the recentness configurations (see Section 3.3.3.4). Also, filtering of management change-sets was added (see Section 3.3.3.5).

- Refinement of the experiment procedures. The number of subject systems was increased to eight and the number of samples increased to 600. A state-of-the-art textual FLT (TFLT_{B}) was implemented to be compared with ACIR. Finally, due to the increased number of samples, the gold set creation was automated, using the gold set builder tool (see Section 4.3 and Section 4.5).

### 4.3 Experiment Design and Work-flow

To run both the pilot and scaled-up studies, a work-flow, as shown in Figure 4.1 was used. First, for each of the subject systems a corpus of indexed documents
(source code entities) was built for each of the ACIR or TFLT$_B$ configurations, as discussed in the previous chapter. The input to these approaches was the source code of the subject systems and their VCS data. Also, for each of these subject systems a gold set (see Section 2.3) was identified. Gold sets were constructed automatically, using a gold set builder, that implemented the re-enactment approach, discussed in Section 2.3. The input to the gold set builder is source code, the VCS, and the ITS. Each gold set served two purposes:

- To construct search queries from feature descriptions.
- To provide the set of “correct” source code entities for each feature in a gold set.

Then for each feature in the gold sets, the result set (the suggested source code entities), that was obtained by querying the corpus of source code entities with the gold set search query, was compared to the correct set (from the gold set) for that feature. Metrics were applied to these sets to quantitatively evaluate the approach (configurations of ACIR or TFLT$_B$) that generated the result set.
4.4 Subject Systems and Their Selection Strategy

The relationship between a subject system and the various sets returned by configurations of approaches is “one-to-many”. The relationship between a subject system and a gold set is “one-to-one”.

The next sections focus on the steps, shown in Figure 4.1 in more detail (except for the ACIR and TFLT approaches which were described in detail in the previous chapter). In particular, the following are discussed:

- How the subject systems were selected.
- How the gold sets were built.
- What metrics were used.
- Replication procedures.

4.4 Subject Systems and Their Selection Strategy

To statistically test the approaches presented in this work and answer the research questions to some degree of generality, eight software systems were selected for the experiment. The intention was to widen the pool of subject systems, compared to the related works, which tended to employ one system only (Zanjani et al., 2014), or a fewer number of very similar systems (Canfora and Cerulo, 2006; Chen et al., 2001). The selected systems were selected based on their adhering to the following requirements:

- **Set of characteristics.** The software systems were of medium size - between 75,000 and 500,000 lines of code. This is because systems need to be of a certain scale for feature location to be valuable: if a developer is dealing with 100-200 lines of code, feature location is not so much of a problem.

1 According to Mayrhauser and Vans (1995) such systems would be classified as large, but in today’s software context would probably be considered as medium.
4.4 Subject Systems and Their Selection Strategy

- **Availability of auxiliary VCS and ITS data.** Because ACIR relies on analysis of change-sets and because ITSs are used to generate gold sets, a subject system has to employ both a VCS and an ITS.

- **Sufficient sample size.** The subject systems were selected so that their total population of change requests was large. For these eight systems there were 11,237 change requests. Based on this number, a sample of at least 566 is needed to conduct statistical tests at 95% confidence level and 4% accuracy range over the population of systems chosen (Oates, 2005). It was decided to take 600 samples, 75 samples for each subject system. An alternative would have been to calculate a sample size based on the number of change requests in each system studied. However, this was infeasible, given that, even in cases which had a vigorous and ongoing history of change requests (Mylyn.Tasks: 533 change requests, JGit: 469 change requests), you would still need a large sample (283 and 264 respectively) for a confidence interval of 95% and an error margin of 4%. These sample-sizes would leave an inadequate number (250 and 232 respectively) of change-sets to annotate the entire code-base.

- **Availability of a software system.** The source code, VCS, and ITS have to be publicly available: hence the system has to be open source or have its source code/VCS/ITS made available to us through its commercial vendor. In this work, the efforts were directed at open source software.

These requirements imply a selection of well-established software systems, with organized communities, and a confirmed history of development. As a rule of thumb, the medium scale open source projects selected have (at least) several years of active development. Also, due to implementation restrictions, only systems that are written in the Java programming language were selected.

As seen in Table 4.2 the subject systems are *Rhino*, a JavaScript engine, *Mylyn.Tasks*, a sub-module of Mylyn task management system, *JGit*, Java library for Git, *Jetty*, Java web container, *Ant*, Java build tool, *Hudson*, continuous integration framework, *JMeter*, web resources performance test application, and

1[http://www.surveysystem.com/sscalc.htm](http://www.surveysystem.com/sscalc.htm)
4.5 Identifying a Gold Set: Re-enactment Approach

Table 4.2: Subject Systems

<table>
<thead>
<tr>
<th>Subject System</th>
<th>Revision</th>
<th>Java LOC (code)</th>
<th># Java Files</th>
<th># Commits (master)</th>
<th># Change requests</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rhino</td>
<td>15f57d57</td>
<td>92899</td>
<td>500</td>
<td>3372</td>
<td>643</td>
</tr>
<tr>
<td>Mylyn.Tasks</td>
<td>d7791d2d</td>
<td>123666</td>
<td>1136</td>
<td>8667</td>
<td>533</td>
</tr>
<tr>
<td>JGit</td>
<td>b601e887</td>
<td>168865</td>
<td>1274</td>
<td>4421</td>
<td>469</td>
</tr>
<tr>
<td>Jetty</td>
<td>970798b8</td>
<td>317108</td>
<td>2556</td>
<td>12215</td>
<td>2741</td>
</tr>
<tr>
<td>Ant</td>
<td>8bb69b6d</td>
<td>136142</td>
<td>1216</td>
<td>13231</td>
<td>2705</td>
</tr>
<tr>
<td>Hudson</td>
<td>43770f72</td>
<td>91450</td>
<td>1066</td>
<td>1555</td>
<td>531</td>
</tr>
<tr>
<td>JMeter</td>
<td>b901505a</td>
<td>119681</td>
<td>1164</td>
<td>12136</td>
<td>1922</td>
</tr>
<tr>
<td>Eclipse.Platform.Text</td>
<td>35e6af06</td>
<td>131835</td>
<td>1083</td>
<td>6273</td>
<td>1693</td>
</tr>
</tbody>
</table>

Eclipse.Platform.Text, a sub-module of the Eclipse project contributing to text editor implementation. All these software systems originated between 1997 and 2011, Rhino and Hudson were the oldest and the most recent respectively. Five of these software systems were previously studied in FL literature ([Liu et al., 2007], [Poshyvanyk et al., 2012], [Würsch et al., 2013], [Zamani et al., 2014]). All of the systems have a (sufficient) history of recorded change requests and strict coding policies. The profile data in Table 4.2 was collected on March 11th 2016 using cloc\footnote{http://cloc.sourceforge.net/} and Git utilities.

4.5 Identifying a Gold Set: Re-enactment Approach

In this work, for each subject system a gold set is constructed from historical changes, recorded in an ITS and a VCS of that subject system, using the automated re-enactment approach (see Section 2.3). A re-enactment approach assumes that a feature location activity could be approximated using the past resolution of a change request. The gold sets, obtained using this approach, are, in fact, collections of change requests that are mapped to source code entities,
that were changed as part of these change requests. The feature descriptions (that are used as search queries) in these gold sets are the change requests’ descriptions. Existing research papers used both short (title) and long (summary) (and their combinations) change requests’ descriptions to formulate the search queries (Gay et al., 2009; Petrenko and Rajlich, 2013; Poshivanyk et al., 2012). Some newer research suggests that more effective results are obtained by formulating the search queries that use only titles (Scanniello et al., 2015). Thus, in this work, it was decided to use short (title) descriptions because the words in the titles are likely to be more carefully/succinctly picked (and hence, more likely to precisely represent a feature) by developers. This seems to be more in line with the real-world scenario where search queries are likely to contain a few, carefully picked, search keywords.

Source code entities that are relevant to a change request can be found by either:

a) Matching the *bug identification number* (bugId) of a change request with those bugIds found in change-set descriptions, related to the change request: those change-sets then lead to source code entities.

b) Inspecting the comments of a change request to find change-sets’ revision numbers: change-sets, having those numbers, then lead to source code entities.

c) Inspecting patches attached to a change request for source code entities.

It was decided to use approach a), because:
4.5 Identifying a Gold Set: Re-enactment Approach

![Diagram of change-set division into annotation and test sets](image)

- It can be automated.
- It was previously used in the literature (Gay et al., 2009; Petrenko and Rajlich, 2013; Poshyvanyk et al., 2012).
- Informal observations of the data-set, performed by the author, suggest support for the intuition that a) returns the most unambiguous answers.

Hence, this approach was adopted in this study and a tool was created to automate the re-enactment. The change requests in the ITS of a subject system were linked to change-sets in the VCS of that subject system via a common bugId. The most recent 75 change requests (the required sample of features to be located) were captured for a total of 600 bug reports across all eight subject systems. All of these eight subject systems used Bugzilla ITS. To interact with Git, the tool used JGit library and to interact with Bugzilla, the tool used B4J library (see Section 3.5).

The approach described above, however, raised a question: how should ACIR treat those change-sets that are part of the change requests in the gold set? Using them for the annotation of source code entities along with other change-sets could be a potential threat to the validity of results: there could be a resultant high degree of similarity between the change requests’ descriptions (search queries) in a gold set and their corresponding change-set descriptions. Removing such change-sets from the annotation of source code entities still could be a threat to the validity of results: in this case one can’t be confident if change-sets aggregation
configurations for a source code entity are valid (e.g., is a source code entity annotated with the most recent change-set descriptions if change-sets, related to a change request are removed?). A more plausible solution is to roll back a software system to a point (change-set), that precedes all of these gold set change-sets.

Such a point (referred to as threshold further in the text) can be found, using the following observation: none of the change-sets related to a change request can occur (and touch the code) before the change request was created (see Figure 4.2). Therefore, the threshold should be selected in such a way that it chronologically precedes the creation time of the oldest (according to its creation time) change request in a gold set as shown in Figure 4.3. In that figure, \( R_k \) is the set of change requests in the gold set, the subject system is rolled back to a threshold \( C_i \), and all the change-sets before and including the threshold are used as an input to ACIR, to annotate source code entities.

However, some of the source code entities from the gold set, might be absent in the source code of a subject system at \( C_i \) (e.g., they were added after \( C_i \)). Hence, it was decided to check that all change requests in the gold set have at least one source code entity that is present in the source code of a subject system at \( C_i \) (so that it is possible to locate such a feature). Otherwise, the change request was removed from the gold set and the next oldest change request was added (this also implies pushing the threshold \( C_i \) back in time). The two latter steps continued until all the change requests in a gold set referred to at least one source code entity.

Finally, to assess the quality of the re-enactment gold-set, two external reviewers were employed to cross-check the approach. Two random samples were extracted from the gold sets used in this work (one from Rhino and one from Mylyn.Tasks), each containing five change requests and their respective source code entities. Each change request was randomly drawn from the pool of 75 change requests used for the scaled up study. For diversity, each change request was checked (based on their descriptions) to ensure it describes different functionality.

For each change request an equal number of randomly selected false candidates (source code entities) were added, along with the “true” source code entities. A
4.6 Metrics

A factor of “closeness” was used when selecting these false candidates: they had to reside in the same package as the actual gold set candidates. In addition, they were often selected to contain part of a variable name/method name/comment that also occurred in the feature description. There were exactly the same number of false candidates for each feature (e.g., three source code entities from the actual gold set and three false candidates).

The reviewers were asked to look at the code (and other parts of source code if they wish) to see which of these source code entities in their opinion should be part of these change requests. They were given a link to the code-base, the feature description and, for each segment of code, its package and the source code itself.

The agreement rate between the two reviewers and the gold set was 84% (they agreed on 42/50 source code entities across 10 change requests). Of these 21 were correctly identified as associated with the change request and the other 21 were correctly identified as not associated with the change request. Such a high agreement rate indicates that the re-enactment approach generates results that are very close to those a human reviewer would deem relevant.

4.6 Metrics

To evaluate the performance of ACIR a set of relevant metrics is needed. According to the objectives of this work both effort metrics and IR metrics are needed.

4.6.1 File/Method Effort Metrics

To determine whether the approach should be applied at method or file level, the effort required of a developer to locate a feature at file/method level of granularity was required. Effort metrics were found in previous works on FL and were extended to suit the needs of the experiment.

1This was done to reduce bias. In effect it helped ensure that the methods like “main” were not compared to the “changeColor” method when locating colour changing functionality.
4.6 Metrics

Petrenko and Rajlich (2013) describe the effort as the number of source code entities a developer has to inspect before reaching the correct entity. In their assessment, a unit of effort is equal to inspection of one particular source code entity (in their case a method). Tantithamthavorn et al. (2014) use lines of code to compare effort required to locate entities at method and file level of granularity. However, a developer would not necessarily read all of the source code of an entity, scanning over lexicons at different levels of granularity to identify relevance (O’Brien et al., 2004). Also, their approach is method-centric: the correct lines of code are located only in methods and it is assumed that a developer is interested in locating methods only. However, what if a developer is looking for a file level entity?

In this work, it was decided to compare the file and method levels of granularity according to how effort intensive each of them would be at locating features at the opposite level of granularity. For example, how effective is ACIR, applied at file level granularity (all source code entities in the IR index are files), in locating features of method level granularity.

For this task, the effort metric, as defined by Petrenko and Rajlich (2013) could be applied. Such effort metrics can be expressed as the “effectiveness metric” (position of the first correct source code entity) and MRR metric (the mean of the inverse of the effectiveness metric across several result sets). But, when the granularities are different, these metrics have to be properly adjusted to facilitate comparison across the two chosen granularities. Two such cases that require proper adjustment are presented below:

• In the first scenario (Case I), a user is searching for source code entities of file level granularity, while the results are presented at method level. Then it is assumed that the first method entry leading to a correct file should be considered the correct answer, whereas multiple method entries leading to the same incorrect file should be counted as one. That is, the effort in this case is equal to the number of unique files a developer has to inspect following the result rankings of all methods $M$ that are ranked before the
first correct method, as in Equation 4.1:

\[ \text{Effort}(\text{Case I}) = \text{unique}(\text{LinksToFiles}(M)) \]  

(4.1)

- Another situation (Case II) is when a user is looking for source code entities of method level granularity, while the results are returned at file level. In this case it is assumed that a user has to inspect all the methods in a file until he/she discovers the correct method. Therefore the effort is then a sum of all methods of all files \( \forall f \in F \) a user has traversed, as prompted by the ranked list in the results, before the correct file is presented in the ranked list, as in Equation 4.2:

\[ \text{Effort}(\text{Case II}) = \sum_{f}^{F} \text{NumberOfMethods}(f) \]  

(4.2)

### 4.6.2 IR metrics

To quantitatively evaluate TFLT\textsubscript{IRS}, researchers often use these metrics (discussed in Section 2.3.3):

- Precision;
- Recall;
- The effectiveness metric;
- MRR;
- AP and MAP.

In this work, the AP and MAP metric were used, because:

- AP effectively combines statistics of both the positions and the number of relevant entities in a ranked list \cite{Baeza-Yates and Ribeiro-Neto 1999}. In contrast, the effectiveness metric and MRR consider only the position of a first correct source code entity, an entity that a user may accidentally miss.
Alternatively, the user may be interested in finding the whole extent of the feature over several entities. Hence, it is important to measure how many other correct entities are close to the top ranked positions.

- Outside FL, AP and MAP are the standard metrics to compare IR systems (and ACIR is an IR-based FLT) \cite{Smucker2007}.

The precision and the recall metrics were not used, because:

- The precision metric was not used, because it is less appropriate in the context of systems that generate ranked lists of results (see Section 2.3.3).

- The recall metric seems more exclusively concerned with finding the extent of a feature, paying less heed to obtaining an entry point to a feature. (It should also be noted that, while AP and MAP do not directly address recall, more correct answers serve to raise the average precision.)

This decision is also in line with other recent studies of TFLTs, where the authors have completely abandoned precision and recall metrics and used the effectiveness metric, MRR, and MAP instead \cite{Bassett2013, Petenko2013, Scanniello2015, Zamani2014}.

4.7 Replication Procedures

This section concludes the chapter by describing the procedures required when applying the ACIR technique to other software systems and building gold sets using the gold set builder tool. This allows for replication of this study, or for exact comparison between ACIR and other approaches, while also allowing researchers generate gold standards matching and beyond the ones employed in this work.

There are four major steps when applying ACIR:

1. Obtain the ACIR implementation. ACIR is available as part of the FUSIX library (see Section 3.5). FUSIX is a Java library and can be included in a Java project via the class-path of the Java project or via dependency management tools (e.g. dependencies in Eclipse).
2. Configure ACIR. Once the implementation has been obtained, ACIR can be configured through the API, as shown in Appendix B: Source Code Example of Using FUSIX API. Particularly the following parameters have to be configured:

- Link to a source directory that specifies where the source code of a software system is located. It is also the link to a Git repository of this software system.
- Link to an index directory that specifies where an IR index is to be stored.
- Granularity has to be selected choosing one of the values described in Section 3.3.3.1.
- Recentness has to be selected choosing one of the values described in Section 3.3.3.1.
- Inclusion/exclusion of management type change-sets has to be specified.
- Revision of the software system has to be specified.

3. Run ACIR. After ACIR is configured it can be launched and will create the search corpus in the index directory.

4. Use ACIR. At this point ACIR is available for FL: a user can submit a search query using the API (see Appendix B: Source Code Example of Using FUSIX API) and the search results will be returned.

---

1 Git repository of a software system usually resides in the source code directory of that same system
5

Evaluation of ACIR

In the previous chapters, the research gap, the novel ACIR approach, and the empirical design used to evaluate the approach were presented to the reader. First in this chapter, the results of a pilot study are presented. Based on the lessons learned from the pilot study, the scaled-up experiment was conducted and is reported further in this chapter. Particularly, research questions of the scaled-up experiment are outlined, followed by presentation of the results of this experiment, discussion of these results, the threats to validity, and the effort data.

5.1 The Pilot Study

In this section, the results of the pilot study, providing a preliminary evaluation of the ACIR approach, are presented. It uses a smaller number of subject systems, answers pilot-study related research questions, and provides an initial, but inexact comparison of ACIR with existing state-of-the-art TFLTs.

5.1.1 The Research Questions of the Pilot Study

The main objective of the pilot study was to get the feel for the effectiveness of ACIR. ACIR is supposed to address the vocabulary paucity problem in TFLT$_{IR}$s by annotating source code entities with change-set descriptions, whereas the existing TFLT$_{IR}$s usually employ textual data in source code to annotate source code entities (Dit et al., 2013a; Marcus and Haiduc, 2013). Hence, there was a
5.1 The Pilot Study

need to assess how effective ACIR is, when compared to these existing approaches and see if this approach is viable, before processing to a scaled-up study. The first research question addresses the effectiveness of ACIR:

**RQ1:** How effective is ACIR with respect to existing textual source code FLTs?

There is an inconsistency between FLT providers as to the granularity of the source code sought by their solutions. Most direct their work at finer level results (mostly method level) \cite{Biggers2014, Cleary2009, Dit2013, Marcus2004, Rajlich2004, Scanniello2011}. However, authors of FLTs rarely justify the selection of one level of granularity over the other. Moreover, there is no explicit evidence in support of this position (choosing a finer level of granularity). Assuming that sometimes programmers will be looking for files, and sometimes for methods, then the key question is whether it is better to present them with files or methods? Ultimately it is about the effort they expend: which consumes less effort to locate features over both common programmer targets: methods and files. Hence, the research question with respect to ACIR is:

**RQ2:** How does the granularity of the source code target (method/file) affect the effort expended by programmers when using ACIR?

The configuration of change-set is unique to meta-data FLTs and, to the author’s knowledge, has not been studied in the FL literature. Still, some interesting conclusions could be drawn from the existing research presented by \cite{Zamani2014} and by \cite{Chen2001}. In the first work the authors make a claim that older terms should receive lower scores when ranking due to the assumption that they become obsolete during the natural life-cycle of a software system (note that, in that work terms were extracted from source code and change-sets’ date meta-data was used to adjust their scores). In contrast, \cite{Chen2001} used all available historical change-sets equally to annotate each line of code for FL. Both works reported the increased effectiveness of their techniques. Though these two approaches are hardly comparable, they lead to an interesting question
5.1 The Pilot Study

Table 5.1: ACIR Descriptive Statistics

<table>
<thead>
<tr>
<th>Subjects System</th>
<th>Granularity</th>
<th>Commit range</th>
<th>Effectiveness</th>
<th>MAP %</th>
<th>MRR %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rhino</td>
<td>Method</td>
<td>Most recent</td>
<td>60</td>
<td>183.43</td>
<td>7.9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>All</td>
<td>75</td>
<td>271.53</td>
<td>6.84</td>
</tr>
<tr>
<td>File</td>
<td>Method</td>
<td>Most recent</td>
<td>4</td>
<td>12.22</td>
<td>39.63</td>
</tr>
<tr>
<td></td>
<td></td>
<td>All</td>
<td>5</td>
<td>18.11</td>
<td>34.27</td>
</tr>
<tr>
<td>Mylyn.Tasks</td>
<td>Method</td>
<td>Most recent</td>
<td>225.5</td>
<td>496.63</td>
<td>10.71</td>
</tr>
<tr>
<td></td>
<td></td>
<td>All</td>
<td>198</td>
<td>422.06</td>
<td>10.99</td>
</tr>
<tr>
<td>File</td>
<td>Method</td>
<td>Most recent</td>
<td>44</td>
<td>62.26</td>
<td>15.97</td>
</tr>
<tr>
<td></td>
<td></td>
<td>All</td>
<td>21</td>
<td>53.84</td>
<td>19.8</td>
</tr>
</tbody>
</table>

about the impact that change-set-range has on the effectiveness of ACIR. The pilot study focuses on the two most extreme points of the change-set range (i.e. the most recent change-sets compared to all change-sets of a source code entity), since they would probably most noticeably highlight any discernible differences. These assumptions lead to the third question:

**RQ3:** How does including only the most recent, as opposed to all, change-sets affect the effectiveness of ACIR?

### 5.1.2 Initial Results and Analysis

In this section, the results of this empirical study are reported and analysed to answer the pilot research questions raised in Section 5.1.1. In Table 5.1 the metrics of four different configurations of ACIR for each subject system are presented. Then, Table 5.2 compares data of existing baseline IR based FLTs from the literature against ACIR. For each configuration the effort, as based on the adjustment scenarios discussed in Section 4.6.1, is calculated and compiled in Table 5.3. The effort values are further analysed in the box-plots of Figure 5.1. Table 5.4 and the box-plots of Figure 5.2 show the effectiveness of ACIR for different ranges of change-set inclusion.
5.1 The Pilot Study

As stated in Section 5.1.1, RQ1 is: *How effective is ACIR with respect to existing textual source code FLT*s? To answer this question, the results between Rhino and Mylyn.Tasks are compared first and then compared against existing FLT*s. When comparing subject systems, the technique showed more effective results when applied to the Rhino sample set (see Table 5.1). That is, for all appropriate settings (i.e. of similar granularity level and change-set range) the effectiveness metric was better for Rhino system. In one case (i.e. the technique configured at a file level of granularity and including the most recent change-sets) the technique resulted in a median of effectiveness metric, that was 10 times better than that of Mylyn.Tasks.

Using the MAP and MRR measures though, the picture is less clear: at the method level of granularity, Mylyn.Tasks showed MAP and MRR to be higher by 60% and 58% respectively in cases when all change-sets were included. However, at a file level of granularity MAP and MRR of Rhino were almost two times higher than those of Mylyn.Tasks. The effectiveness metric, the MAP, and the MRR are poorer for both systems at a method level granularity. This is because documents of a file level granularity will likely have more matching terms against a search query, which in turn will increase their cosine similarity.

To compare ACIR against existing FLT*s the results reported in the FL literature were gathered and compiled into Table 5.2. Rigid criteria were applied when selecting the approaches for comparison. They had to:

- Employ source code data for textual FL.
- Use comparable IR model such as VSM or LSI along with TF-IDF scoring function.
- Be applied at file/method level granularity.
- Use subject systems written in Java for evaluation.
- Report comparable metrics such as the effectiveness metric, MAP, and MRR.
Six studies were found that match these criteria (Dit et al., 2011, 2013b; Petrenko and Rajlich, 2013; Scanniello and Marcus, 2011; Sisman and Kak, 2013; Zamani et al., 2014). Four of these techniques were drawn from a population of 27 research papers identified during the literature review of textual FLTs that include empirical studies and that were carried out between 2011 and 2015 (see Section 2.2.2.1). The paper by Dit et al. (2013b) does not describe a novel FLT, but reports an empirical study. It also matched all of the above criteria and because of these factors was included in this list. The paper by Sisman and Kak (2013) describes a BL approach (that does not use change-set descriptions). However, because of the high similarity between BL and FL approaches and because this paper matched all the above criteria it was included in this list. Of those six approaches, three used Rhino as a subject system (Dit et al., 2011, 2013b; Zamani et al., 2014), whereas the rest reported on other Java projects (Petrenko and Rajlich, 2013; Scanniello and Marcus, 2011; Sisman and Kak, 2013). Those approaches that studied Rhino are considered to be slightly more relevant in terms of comparison.

In Table 5.2, the parameters of each approach are reported only if authors state them explicitly, otherwise it is assumed that baseline IR using VSM and TF-IDF function is employed. In those cases where the analysis had to rely on the box-plot data (Dit et al., 2011, 2013b), the metrics are presented as a range of values. Along with the results of these six approaches, the corresponding metrics of ACIR, derived from the data shown in Table 5.1 are reported and highlighted in bold if they show better results. The first number inside the parentheses is the average of the metric at a given granularity and the second number is the value of the best performing configuration at that same granularity. In case of non matching subject systems the average of the appropriate metrics for both Rhino and Mylyn was derived. As can be seen from Table 5.2 the median of the effectiveness metric of ACIR was better in four cases out of 5. The mean of the effectiveness metric was better in two cases out of five when the average effectiveness metric was taken and in three cases out of five when the effectiveness metric of the best configuration was taken. The MAP and the MRR of ACIR were better for all reported approaches (in some cases the MAP was significantly better,
Table 5.2: Comparing ACIR with Previous Work

<table>
<thead>
<tr>
<th>Technique</th>
<th>Subject System</th>
<th>Granularity</th>
<th>IR model</th>
<th>Score function</th>
<th>Effectiveness</th>
<th>MAP %</th>
<th>MRR %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matching subject systems</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zamani'14 Zamani et al. (2014)</td>
<td>Rhino File</td>
<td>VSM</td>
<td>TF*IDF</td>
<td>5</td>
<td>8.6</td>
<td>2.89</td>
<td>38.0</td>
</tr>
<tr>
<td>Dit'12 Dit et al. (2013b)</td>
<td>Rhino Method</td>
<td>LSI</td>
<td></td>
<td>(4.5; 4)</td>
<td>(15.17; 12.22)</td>
<td>(36.95; 39.63)</td>
<td>(44.07; 44.32)</td>
</tr>
<tr>
<td>Dit'11 Dit et al. (2011)</td>
<td>Rhino Method</td>
<td>LSI</td>
<td></td>
<td>(67.5; 60)</td>
<td>(227.48; 183)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Non matching subject systems</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sisman'13 Sisman and Kak (2013)</td>
<td>Eclipse Chrome</td>
<td>TF*IDF</td>
<td></td>
<td>20.89</td>
<td>15.35</td>
<td>(27.42; 29.72)</td>
<td></td>
</tr>
<tr>
<td>Scanniello'11 Scanniello and Marcus (2011)</td>
<td>Eclipse</td>
<td>Method</td>
<td>VSM</td>
<td>TF*IDF</td>
<td>217</td>
<td>630</td>
<td>(139.63; 129)</td>
</tr>
<tr>
<td>Petrenko'13 Petrenko and Rajlich (2013)</td>
<td>Adempiere DrJava JabRef jEdit</td>
<td>Method</td>
<td></td>
<td>10</td>
<td>190</td>
<td>(139.63; 129)</td>
<td>(343.41; 302.75)</td>
</tr>
</tbody>
</table>
5.1 The Pilot Study

Table 5.3: Effort for File vs Method Level Cases of Granularity

<table>
<thead>
<tr>
<th>Subject Systems</th>
<th>Granularity levels</th>
<th>Average Effort</th>
<th>Effort change %</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>FLr vs MLr(a)</td>
<td>12.22</td>
<td>+19.34</td>
</tr>
<tr>
<td>Rhino</td>
<td>FLh vs MLh(a)</td>
<td>18.11</td>
<td>-24.3</td>
</tr>
<tr>
<td>Mylyn.Tasks</td>
<td>FLr vs MLr(a)</td>
<td>62.26</td>
<td>-30.85</td>
</tr>
<tr>
<td></td>
<td>FLh vs MLh(a)</td>
<td>53.84</td>
<td>-6.26</td>
</tr>
<tr>
<td>Case II</td>
<td>FLr(a) vs MLr</td>
<td>519.67</td>
<td>-64.70</td>
</tr>
<tr>
<td>Rhino</td>
<td>FLh(a) vs MLh</td>
<td>646.22</td>
<td>-57.98</td>
</tr>
<tr>
<td>Mylyn.Tasks</td>
<td>FLr(a) vs MLr</td>
<td>905.84</td>
<td>-45.17</td>
</tr>
<tr>
<td></td>
<td>FLh(a) vs MLh</td>
<td>752.32</td>
<td>-43.9</td>
</tr>
</tbody>
</table>

The legend to read the data: FL - file level, ML - method level, r - recent, h - all historical, a - adjusted, as per Section 4.6.1. This preliminary data on this small dataset suggests, that the technique is at least competitive and shows effective performance.

To answer RQ2 ‘How does the granularity of the source code target (method-/file) affect the effort expended by programmers when using ACIR?’, the effectiveness metric (effort) had to be adjusted for two cases (Case I and Case II) as discussed in Section 4.6.1. The observed effectiveness metric statistics are significantly better when the technique is applied at file level granularity (see Table 5.1). Based on the effectiveness statistic, the effort was calculated and the results are shown in Table 5.3. In this table, the two “Average Effort” columns refer to the two approaches listed in the “Granularity levels” columns respectively. So, for example, in row one, 12.22 refers to the FLr approach and 14.59 refers to the MLr(a) approach. According to the data in the table, the technique, applied at method level of granularity, reduces the effort by up to 31% for Case I and by up
5.1 The Pilot Study

Figure 5.1: The effort of different levels of granularity: a) when the method level effectiveness is adjusted; b) when the file level effectiveness is adjusted.

* The legend to read the data: FL - file level, ML - method level, r - recent, h - all historical, a - adjusted, R - Rhino, M - Mylyn.Tasks.

to 65% for Case II. In one case $FLr$ vs $MLr(a)$ the effort was increased by 19%, though the difference in absolute numbers was not that significant (i.e. 12.22 vs 14.59 respectively).

Figure 5.1a and Figure 5.1b visualize the effort data for cases I and II. As could be seen from Figure 5.1a for two cases $FLh$-vs-$MLh(a)$ and $FLr$-vs-$MLr(a)$ there is an obvious difference in the data distribution of box-plots (i.e. group two and 3) that further supports the initial observation that less effort is required when the technique is configured at a method level of granularity. The difference is not that obvious for the last group of box-plots ($FLh$ vs $MLh(a)$). Still the interquartile range (IQR), the upper quartile, and the upper whisker were slightly lower at method level of granularity. For all the cases in Figure 5.1b there was an obvious difference in the box-plot data. Summarizing, there is initial support in this preliminary data-set for decreased effort when the technique is configured at method level granularity for at least six cases shown in Table 5.3.

One possible explanation for this effect could come from the statistics of Ta-
5.1 The Pilot Study

Table 5.4: Effectiveness for All Historical vs Recent Cases of change-set Range

<table>
<thead>
<tr>
<th>Subject Systems</th>
<th>Granularity</th>
<th>Effectiveness Historical</th>
<th>Effectiveness Recent</th>
<th>Eff. change for recent commits %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rhino</td>
<td>File</td>
<td>18.11</td>
<td>12.22</td>
<td>+32.52</td>
</tr>
<tr>
<td>Rhino</td>
<td>Method</td>
<td>271.53</td>
<td>183.43</td>
<td>+32.45</td>
</tr>
<tr>
<td>Mylyn.Tasks</td>
<td>File</td>
<td>53.84</td>
<td>62.26</td>
<td>-15.64</td>
</tr>
<tr>
<td>Mylyn.Tasks</td>
<td>Method</td>
<td>422.06</td>
<td>496.63</td>
<td>-17.67</td>
</tr>
</tbody>
</table>

Table 4.1. There are 7.84 and 13.07 methods per file on the average for Mylyn.Tasks and Rhino respectively that add to the effort required to reach the source code entities at method level. That is, this finding depends on the definition of the effort that was established in Section 4.6.1. Regardless, the answer to RQ2, based on this initial study and the effort definition used in this work, is that the technique, configured at method level granularity decreases effort, for these two systems.

To answer RQ3 How does including only the most recent, as opposed to all, change-sets affect the effectiveness of ACIR? the impact of change-set range inclusion was considered. The average effectiveness of all configurations of Rhino system reported in Table 5.1 shows that the inclusion of only the most recent change-sets produces more efficient results. However, for Mylyn.Tasks the most efficient configurations utilized all historical change-sets. The results for Rhino show that, when using the most recent change-sets, the effectiveness has improved (recall that lower effectiveness is better - see Section 4.6.1) by up to 32% as presented in Table 5.4, whereas including the most recent change-sets in Mylyn.Tasks results in the effectiveness decreased by up to 17%. A look at the Rhino box-plots in Figure 5.2a shows that there is more obvious difference in data distribution, than in case of Mylyn.Tasks, as seen in Figure 5.2b. A difference in the observable effect could be explained if the development history of subject systems is analysed. Rhino is a fairly mature project tracing its origins back to 1997 (see Table 4.2). During that time the vocabulary used to describe the program concepts may have considerably evolved rendering change-set descriptions of older
5.1 The Pilot Study

Figure 5.2: The effectiveness of different change-set ranges: a) in Rhino; b) in Mylyn.Tasks.

\* The legend to read the data: FL - file level, ML - method level, r - recent, h - all historical, a - adjusted.

change-sets obsolete. Mylyn.Tasks is almost eight years younger and most likely has retained more of its original concepts to date. Addressing RQ3 then, it seems that employing recent change sets only may change the performance of the technique but that this may be system dependent and may be dependent on the age of the system.

In summary, the preliminary findings presented here suggest that ACIR is effective when compared to existing TFLT\textsubscript{IR}s. For the two observed subject systems it was found, that ACIR configured at a method level of granularity allows an effort reduction of up to 64%. The preliminary results regarding the inclusion of recent change-sets only are inconclusive, but may be dependent on the subject system.

5.1.3 Initial Conclusions

In this pilot study a newly proposed FLT, ACIR, that explores fitness of change-set descriptions as an alternative data source for TFLT\textsubscript{IR}s underwent preliminary
5.2 Objectives, Research Questions, and Hypotheses of the Scaled-Up Empirical Study

The following objectives were set for this study:

- To identify the best performing configuration of ACIR.
- To identify how the best performing configuration of ACIR compares to TFLT\textsubscript{B}.

Consequently, five research questions were defined for this empirical study:

**RQ1:** What effect does granularity of target source code entities have on developers’ effort? This first question studies if there is an advantage to using file or method level granularity with the ACIR approach. Only one study has been performed in this space before \cite{Tantithamthavorn et al., 2014}.
5.2 Objectives, Research Questions, and Hypotheses of the Scaled-Up Empirical Study

It indicated that method-level granularity is more advantageous when locating functionality using source code lexicons as a data source. Here it is assessed if the same principle applies to change-set descriptions data. Essentially the question asks if method-level granularity demands less effort than file-level when trying to locate the first source code entity relevant to a feature.

RQ2: What is the effect of recentness of change-sets? Previously in research that leveraged change set information, only all change-sets were aggregated (Canfora and Cerulo 2006; Chen et al. 2001; Zanjani et al. 2014), but this assumption is questioned in this research.

RQ3: What is the effect of aggregating recent change-sets by change request? This question asks if aggregation of recent change-sets by their change requests, as opposed to just using the latest change-sets that touch the code, improves ACIR. Literature review suggests that such an approach has never been studied.

RQ4: What is the effect of filtering management change-sets? This question studies if there is an advantage in filtering out management change-sets for ACIR. It is based on the assumption that management change-sets may introduce noise and decrease performance, because they reflect non-functional change while often touching (and thus annotating) large sways of the code base.

RQ5: How does ACIR compare to TFLT\textsubscript{B}? This research question compares the best-configuration (as defined in research questions 1-4) ACIR approach to TFLT\textsubscript{B}. Particularly, the distribution of correct results returned by these two techniques is assessed.

For each of the research questions, null hypotheses were formulated. Since it wasn’t known in advance which of the techniques/configurations, if any, would outperform the other, the two-sided hypotheses were used. Below is a list of the null hypotheses associated with research questions:
5.3 Results of the Scaled-Up Study

**H₀₁**: Levels of granularity in ACIR do not introduce any significant difference in effort, when finding the first relevant document.

**H₀₂**: Recentness of change-sets does not introduce any significant difference in the distribution of the correct answers for ACIR.

**H₀₃**: Aggregation of recent change-sets by change request does not introduce any significant difference in the distribution of the correct answers for ACIR.

**H₀₄**: Filtering of change-sets in ACIR does not introduce any significant difference in the distribution of correct answers for ACIR.

**H₀₅**: There is no difference in the distribution of correct answers for ACIR and TFLT₂.

5.3 Results of the Scaled-Up Study

To answer the research questions of this empirical study, the empirical design (detailed in Chapter 4) was employed. Additionally, four configurations of ACIR (all historical change-sets ACIRₐ, most recent change-sets only ACIRᵣ, aggregated recent change-sets by change request ACIRᵣₖ, and most recent filtered change-sets ACIRᵣₕ) were applied. These configurations were used with the test sets of each of the eight subject systems at file and method level of granularity and their performance was measured using the MAP metric. To answer RQ5, the TFLT₂ approach was compared to the best-configuration ACIR, also using the MAP metric. In this evaluation the list of returned results was not limited, a usual practice when evaluating IR systems [Smucker et al., 2007]. Descriptive characteristics of MAP for all settings of ACIR and TFLT₂ are provided in Table 5.5.

The following strategy was employed to systematically analyse results. First, configurations of ACIR are studied to see if programmer effort differs when the technique is applied at file and method level. Subsequently, the best-performing configuration of ACIR, in terms of recentness, and filtering was determined. The best configuration of ACIR was then compared to the state-of-the-art baseline approach TFLT₂ (based on source code lexicons).
5.3 Results of the Scaled-Up Study

Figure 5.3: Distribution of average precision data. x-scale - average precision, y-scale - density.
5.3 Results of the Scaled-Up Study

<table>
<thead>
<tr>
<th>Subject system</th>
<th>Rhino</th>
<th>Mylyn.Tasks</th>
<th>JGit</th>
<th>Jetty</th>
<th>Ant</th>
<th>Hudson</th>
<th>Jmeter</th>
<th>Eclipse Platform.Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method level</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ACIR_A</td>
<td>2.5</td>
<td>12.97</td>
<td>11.95</td>
<td>18.96</td>
<td>2.53</td>
<td>1.29</td>
<td>6.01</td>
<td>3.49</td>
</tr>
<tr>
<td>ACIR_R</td>
<td>10.48</td>
<td>40.08</td>
<td>35.77</td>
<td>54.05</td>
<td>4.59</td>
<td>4.85</td>
<td>8.33</td>
<td>7.98</td>
</tr>
<tr>
<td>ACIR_RC</td>
<td>8.61</td>
<td>38.63</td>
<td>35.00</td>
<td>50.92</td>
<td>4.00</td>
<td>4.06</td>
<td>7.81</td>
<td>7.97</td>
</tr>
<tr>
<td>ACIR_RF</td>
<td>9.82</td>
<td>32.12</td>
<td>32.68</td>
<td>48.89</td>
<td>4.56</td>
<td>5.00</td>
<td>8.22</td>
<td>7.14</td>
</tr>
<tr>
<td>File level</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ACIR_A</td>
<td>36.65</td>
<td>47.77</td>
<td>46.20</td>
<td>56.85</td>
<td>28.75</td>
<td>14.73</td>
<td>31.89</td>
<td>33.91</td>
</tr>
<tr>
<td>ACIR_R</td>
<td>34.42</td>
<td>51.23</td>
<td>44.48</td>
<td>61.86</td>
<td>25.58</td>
<td>16.97</td>
<td>33.49</td>
<td>36.95</td>
</tr>
<tr>
<td>ACIR_RC</td>
<td>35.47</td>
<td>47.91</td>
<td>45.15</td>
<td>59.57</td>
<td>24.03</td>
<td>15.04</td>
<td>32.9</td>
<td>35.68</td>
</tr>
<tr>
<td>ACIR_RF</td>
<td>37.54</td>
<td>40.63</td>
<td>45.67</td>
<td>57.13</td>
<td>25.51</td>
<td>17.73</td>
<td>32.2</td>
<td>35.90</td>
</tr>
<tr>
<td>TFLT_B</td>
<td>35.63</td>
<td>32.64</td>
<td>36.26</td>
<td>33.87</td>
<td>36.91</td>
<td>28.07</td>
<td>38.25</td>
<td>34.97</td>
</tr>
</tbody>
</table>

Statistical testing is used to check the hypotheses presented in Section 5.2. Statistical tests require a data to be distributed in a certain way (e.g. a T-test requires a normal distribution of data). Therefore, first, normality of data distribution was checked for average precision values of all subject systems when applying various ACIR configurations and TFLT_B using Shapiro-Wilk’s test (it was previously used by other FL researchers to check for normality of data distribution, for example by Zamani et al. (2014)). As seen from Figure 5.3, the data seems to be non normally distributed peaking at extreme values. The Shapiro-Wilk’s test showed p-values below 2.2e-16 for all ACIR configurations and TFLT_B, which strongly confirms non normality of the data distribution. Therefore, to conduct statistical analysis testing, the non-parametric Wilcoxon signed rank test was used. This test is suitable for analysis of paired samples that do not follow normal distribution and has previously been used in FL literature and in IR in general (Poshyvanyk et al., 2012; Smucker et al., 2007; Zamani et al., 2014).
5.3 Results of the Scaled-Up Study

Table 5.6: The effort required when finding a file and method expressed in MRR(%) along with Wilcoxon significance test

<table>
<thead>
<tr>
<th>ACIR approaches</th>
<th>Effort of finding a file (MRR %)</th>
<th>Effort of finding a method (MRR %)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Method Level</td>
<td>File Level</td>
</tr>
<tr>
<td></td>
<td>(Case I)</td>
<td></td>
</tr>
<tr>
<td>ACIR_A</td>
<td>43.89</td>
<td>42.40</td>
</tr>
<tr>
<td></td>
<td>10.53</td>
<td>9.31</td>
</tr>
<tr>
<td>ACIR_R</td>
<td>43.59</td>
<td>43.86</td>
</tr>
<tr>
<td></td>
<td>28.52</td>
<td>9.53</td>
</tr>
<tr>
<td>ACIR_RC</td>
<td>45.70</td>
<td>42.96</td>
</tr>
<tr>
<td></td>
<td>26.48</td>
<td>9.42</td>
</tr>
<tr>
<td>ACIR_RF</td>
<td>42.79</td>
<td>43.24</td>
</tr>
<tr>
<td></td>
<td>28.32</td>
<td>8.38</td>
</tr>
</tbody>
</table>

5.3.1 The Effects of Granularity to Effort

The effort which is required off a developer to locate source code entities at file and method level granularity under four configurations of ACIR was measured. These configurations were: ACIR_A, ACIR_R, ACIR_RC, and ACIR_RF. Each configuration was applied to eight subject systems, for a total of 600 change request samples, and the effort was calculated for each sample for Cases I (when the user is looking for files) and II (when the user is looking for methods, see Section 4.6.1). The adapted MRR metric (as reported in Section 4.6.1) was used to conveniently represent the effort across eight subject systems. Wilcoxon statistical testing was then used to assess the significance of differences between the effort data of working at method and file level granularity when trying to find a file (Case I), and between the effort data of working at method level and file level granularity when trying to find a method (Case II).

As shown in the left hand side of Table 5.6, when looking for a file level entity, ACIR_R and ACIR_RF were marginally better when used at file level granularity as measured by the effort metric. In the case of ACIR_A and ACIR_RC a somewhat more substantial, and statistically significant improvement, was observed at method level granularity. Still, the improvements for any of these configurations of ACIR do not exceed 6.37%.

A different picture was observed when a user is looking for an entity at the method level of granularity, as shown on the right hand side of Table 5.6. In this case all of the configurations of ACIR demonstrated significant decrease in effort (as indicated by increase in MRR) of up to 238% when ACIR is indexed
5.3 Results of the Scaled-Up Study

Table 5.7: Wilcoxon signed rank statistical test of MAP gains for ACIR_A and ACIR_R

<table>
<thead>
<tr>
<th>Descriptive MAP statistics</th>
<th>Approach</th>
<th>N</th>
<th>Mean</th>
<th>StDev</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ACIR_A</td>
<td>8</td>
<td>7.4625</td>
<td>6.40991</td>
<td>1.29</td>
<td>18.96</td>
</tr>
<tr>
<td></td>
<td>ACIR_R</td>
<td>8</td>
<td>20.7663</td>
<td>19.43816</td>
<td>4.59</td>
<td>54.05</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Wilcoxon signed rank significance test of MAP</th>
<th>Comparison</th>
<th>Mean (ACIR_A)</th>
<th>Mean (ACIR_R)</th>
<th>Increase</th>
<th>p-value</th>
<th>Significant @0.05</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACIR_A vs. ACIR_R</td>
<td>7.4625</td>
<td>20.7663</td>
<td>+178</td>
<td>0.01172</td>
<td>Yes</td>
<td></td>
</tr>
</tbody>
</table>

at method level of granularity. Statistical testing further supports these results with strongly significant results for all these findings.

To sum up, there is very small difference in terms of effort, and mixed results as to which level of granularity to use, when locating file level entities. But when locating a method, method level granularity substantially outperforms that of file level granularity for all ACIR configurations. Therefore, while H_0 cannot be rejected, it seems advantageous to use method level granularity to search for a method and not harmful to use method level granularity when searching for a file. Consequently, it seems that method level of granularity is more appropriate to use by default. Using this observation, further analysis was limited to method level, for the remainder of this experiment.

5.3.2 The Effects of Recentness

Revisiting Table 5.5, at method level of granularity for all subject systems an increase in MAP was observed when moving from ACIR_A to ACIR_R. To evaluate these observations, the MAPs of the eight subject systems were compared and tested to see if the differences were significant (see Table 5.7). There is strong evidence obtained that this data comes from different distributions and that this difference is statistically significant with a low p-value of 0.01172. As can be seen, the efficiency of ACIR_R increases by 178% when compared to ACIR_A.
5.3 Results of the Scaled-Up Study

To summarize, the evidence obtained allows the repudiation of $H_{o2}$. ACIR$_R$ produces better results at method level of granularity for the systems studied. Hence, at this level of granularity, recentness of change-sets has a strong impact on the MAP of ACIR. Indeed, as seen from Table 5.5, ACIR$_R$ was better than ACIR$_A$ for all subject systems.

5.3.3 The Effects of Aggregating Recent Change-sets by Change Request

Table 5.8: Wilcoxon signed rank statistical test of the effects of aggregation of recent change-sets by change request

<table>
<thead>
<tr>
<th>Approach</th>
<th>N</th>
<th>Mean</th>
<th>StDev</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACIR$_R$</td>
<td>8</td>
<td>20.7663</td>
<td>19.43816</td>
<td>4.59</td>
<td>54.05</td>
</tr>
<tr>
<td>ACIR$_{RC}$</td>
<td>8</td>
<td>19.6300</td>
<td>18.74859</td>
<td>4.00</td>
<td>50.92</td>
</tr>
</tbody>
</table>

Wilcoxon signed rank significance test of MAP

<table>
<thead>
<tr>
<th>Comparison</th>
<th>Mean (ACIR$_{RC}$)</th>
<th>Mean (ACIR$_R$)</th>
<th>Increase</th>
<th>p-value</th>
<th>Significant @0.05</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACIR$_{RC}$ vs. ACIR$_R$</td>
<td>19.6300</td>
<td>20.7663</td>
<td>+5.79</td>
<td>0.01729</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The study of the effects of aggregation of recent change-sets by change request is summarized in Table 5.8. The table provides a summary of the descriptive statistics for the MAP of the eight subject systems. As can be seen, for the entire population ACIR$_R$ was 5.79% better than ACIR$_{RC}$. Though the difference does not seem huge, statistical testing demonstrates strong evidence that this difference is significant across the data-set.

Hence, for this sample, at method level, $H_{o3}$ can be rejected. Surprisingly, ACIR$_R$ proved to produce better results at method level of granularity than ACIR$_{RC}$ for the systems studied.
Table 5.9: Wilcoxon signed rank statistical test of filtering effects

<table>
<thead>
<tr>
<th>Approach</th>
<th>N</th>
<th>Mean</th>
<th>StDev</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACIR</td>
<td>8</td>
<td>20.7663</td>
<td>19.43816</td>
<td>4.59</td>
<td>54.05</td>
</tr>
<tr>
<td>ACIRRF</td>
<td>8</td>
<td>18.5501</td>
<td>16.89029</td>
<td>4.56</td>
<td>48.89</td>
</tr>
</tbody>
</table>

Wilcoxon signed rank significance test of MAP

<table>
<thead>
<tr>
<th>Comparison</th>
<th>Mean (ACIRRF)</th>
<th>Mean (ACIRR)</th>
<th>Increase %</th>
<th>p-value</th>
<th>Significant at 0.05</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACIRRF vs. ACIRR</td>
<td>18.5501</td>
<td>20.7663</td>
<td>+12</td>
<td>0.03569</td>
<td>Yes</td>
</tr>
</tbody>
</table>

5.3.4 The Effects of Filtering Management Change-sets

The study of the effects of filtering is summarized in Table 5.9, where the summary of descriptive statistics for the MAP of the eight subject systems is provided. Statistical testing demonstrates strong evidence of difference between the distribution of results when ACIRR and ACIRRF are compared: as can be seen, for the entire population ACIRR was 12% better than ACIRRF.

Hence, for this sample, at method level, $H_0^4$ can be rejected. Surprisingly, it seems that simple filtering of management change-sets, based on naive text classification, detrimentally impacts the efficiency of ACIR.

5.3.5 TFLT_B vs ACIR_R

Based on the previous sections, the best performing configuration seems to be ACIRR. To answer LRQ5, this configuration was compared with TFLT_B. The results are compiled into Table 5.10.

As illustrated in the table, an interesting dichotomy was discovered. Half of the subject systems, comprising of Rhino, Mylyn.Tasks, JGit, and Jetty (Group A) demonstrated increased MAP when ACIRR was applied. The other half of the subject systems, comprising of Ant, Hudson, Jmeter, and Eclipse.Platform.Text (Group B) showed increased MAP when TFLT_B was used. Since a sample of four MAP values is too small for statistical testing, the average precision data for each
5.4 Discussion of the Scaled-Up Study Results

Table 5.10: Wilcoxon signed rank statistical test of MAP gains for eight subject systems

<table>
<thead>
<tr>
<th>Subject System</th>
<th>MAP (%)</th>
<th>TFLT_B</th>
<th>ACIR_R</th>
<th>p-value</th>
<th>Significant @ 0.05</th>
</tr>
</thead>
<tbody>
<tr>
<td>Group A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rhino</td>
<td>9.82</td>
<td>10.48</td>
<td>+6.72</td>
<td>0.5994</td>
<td>No</td>
</tr>
<tr>
<td>Mylyn.Tasks</td>
<td>18.41</td>
<td>40.08</td>
<td>+117.71</td>
<td>2.669e-05</td>
<td>Yes</td>
</tr>
<tr>
<td>JGit</td>
<td>16.05</td>
<td>35.77</td>
<td>+122.87</td>
<td>7.886e-05</td>
<td>Yes</td>
</tr>
<tr>
<td>Jetty</td>
<td>13.63</td>
<td>54.05</td>
<td>+296.55</td>
<td>1.428e-10</td>
<td>Yes</td>
</tr>
<tr>
<td>Average</td>
<td>14.48</td>
<td>35.09</td>
<td>+142.33</td>
<td>7.215e-15</td>
<td>Yes</td>
</tr>
<tr>
<td>Group B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ant</td>
<td>8.47</td>
<td>4.59</td>
<td>+84.53</td>
<td>0.0131</td>
<td>Yes</td>
</tr>
<tr>
<td>Hudson</td>
<td>10.61</td>
<td>4.85</td>
<td>+118.76</td>
<td>0.005568</td>
<td>Yes</td>
</tr>
<tr>
<td>JMeter</td>
<td>14.22</td>
<td>8.33</td>
<td>+70.71</td>
<td>0.1241</td>
<td>No</td>
</tr>
<tr>
<td>Eclipse.</td>
<td>9.12</td>
<td>7.98</td>
<td>+14.29</td>
<td>0.1471</td>
<td>No</td>
</tr>
<tr>
<td>Platform.Text</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Average</td>
<td>10.61</td>
<td>6.44</td>
<td>+64.75</td>
<td>5.369e-05</td>
<td>Yes</td>
</tr>
</tbody>
</table>

A subject system was used to assess each system individually and an aggregation of these values across all systems in a group was used to assess them in general.

The statistical testing of the total average precision data taken for each group supports significance of the differences between ACIR_R and TFLT_B in groups A and B. Subject systems of Group A, demonstrated a sharper increase in MAP: up to 296.55% for some subject systems when using ACIR_R and a 142.33% increase on average. Group B members showed a slightly lower difference of up to 118.76% when using TFLT_B and, on average, an increase of 64.75%. Obviously, across the eight systems, H_05 cannot be rejected, given the observed dichotomy.

5.4 Discussion of the Scaled-Up Study Results

In this section the results of the study are analysed and explained: first the effects of granularity, then recentness, then type, and then the comparison of ACIR to
5.4 Discussion of the Scaled-Up Study Results

TFLT_B.

5.4.1 The Effects of Granularity for Effort in ACIR

In FL, the possible range of granularity of source code entities varies from very fine level (the finest being the line of code) to extremely large, such as packages in languages like Java. Some levels of granularity are programming language independent (for example line of code and file level), some are common to almost all programming languages (for example method/function level), and some are more language-specific (for example class, package, and structure). The method and file level granularities seem to be the most popular choices in FLTs (Dit et al., 2013a). In fact, existing FLTs seem to prefer method level granularity over the file level. For example, Marcus and Haïduc (2013) cite 24 FLTs that use method/function level granularity and just three FLTs that use file level granularity for FL. The authors give three reasons for such obvious preference for method level granularity:

1. More accurate mapping of features.

2. Easy integration with static and dynamic approaches (though it seems to be as easy to get a file trace as a method trace).

3. Unofficial choice influenced by the need to compare to previous studies.

While points 2) and 3) suggest how method level granularity became a dominating choice in FL, it hardly justifies its selection over the file level granularity in terms of FL effectiveness. Point 1) suggests that the finer level of granularity, when method level granularity is used, translates into increased accuracy, which, in turn, is reflected in increased effectiveness of FL approaches. However, this intuition was not confirmed by experimental evaluation, until recently. This means that, previously the selection of method level granularity was mostly driven by intuition and established conventions in FL.

Marcus and Haïduc (2013) note that the choice of granularity can greatly influence the results of FL technique. For example, in TFLTs particularly, the granularity could influence the number of terms, extracted to describe a source
code entity, and this could influence the efficiency of TFLT_{IR}. More recently, the study by Tantithamthavorn et al. (2014) attempted to objectively evaluate the impact of level of granularity at class and function level. In their work, the effort required to locate source code entities at both class and function (sub-program) level was compared. To compare the entities of different granularity the cumulative number of lines of code that belong to relevant source code entities (functions or classes) in the ranked list of results was used. For example, an inspection threshold could be set to 5,000 lines of code and there could be 100 relevant lines of code at function level granularity and 500 at class level. The number of issues for which the source code entities could be located (there is at least one relevant line of code above the threshold) using such approach was then calculated and the effort metric was defined as the proportion of change requests, for which entities could be located, using these two granularity levels. The major finding was that almost seven times less effort is required to locate features, when the TFLT is using function level granularity.

In this work, the navigation effort of FL was compared when file and method levels of granularity are used in ACIR. The experiment was different from that of Tantithamthavorn et al. (2014) in several ways:

- The approach used in this work is less reductionist/binary: here it is not a distinction between found/not found within a threshold, but instead focuses on the navigation effort to find relevant code.

- Their approach is sub-program centric: correct source code entities are sub-programs only. The approach used in this work assumes a developer is interested in either files or sub-programs.

- This work was focused on the granularity issue for change-set description annotation, not using source code identifiers/comments for creating the bag-of-words.

It was found that when locating method level entities, the ACIR configuration of method level granularity is more advantageous in terms of effort and requires almost three times less effort in some cases. At file level granularity
no significant difference was found. Comparing results of this study and that of Tantithamthavorn et al. (2014), the following conclusions could be drawn:

- The method/function level granularity is more efficient in terms of effort when feature locating.

- This difference in effort seems to be more pronounced for source code TFLTs than for ACIR, which uses change-set descriptions as a data source.

These conclusions naturally raise several questions: ‘Why does the method level of granularity requires less effort to feature locate using TFLTs?’ and ‘Why do the results for ACIR seem to be less pronounced?’.

In TFLTs, the lesser level of granularity means that a source code entity (its textual representation) is less verbose. However, as shown above, this lower verbosity seems to be more advantageous in terms of effort required for FL using TFLTs. One explanation is that file/class level entities will usually contain higher numbers of cross-cutting concerns/features. For TFLT\textsubscript{IR}, this translates into a higher amount of noisy textual data at file/class level granularity and less accurate retrieval of entities. This translates into higher amounts of effort required when locating features at that level. The explanation to the second question could be derived from a similar observation: exactly the same change-set descriptions could annotate several source code entities of method level granularity, even across files. Hence, these source code entities will have more similar textual representation, decreasing the difference in method/file level effort.

Finally, until now all the FLTs described in the literature use granularity levels that are derived from programming language structures or file system structures (e.g. line of code, method, class, file). Obviously, all these structures could be a part of a feature (hence could be used for FL) and the method level granularity seems to be the most advantageous of all of them. However, this raises a question ‘How optimal is method-level granularity?’ Could it be more effective to use more ‘feature-like’ granularity levels, that go beyond programming language and file system structures and reflect the feature itself rather than the structure? For example, a multi-level granularity (file, sub-program, line of code) that partitions source code into a set of frequently co-changing pieces, regardless of granularity, could be studied.
5.4 Discussion of the Scaled-Up Study Results

5.4.2 The Effects of Change-sets Recentness: Evolving Vocabulary

In the past, the techniques in FL and related fields, that used change-sets’ textual descriptions, aggregated all available change-sets (Canfora and Cerulo, 2006; Chen et al., 2001; Zanjani et al., 2014). Such a strategy could be justified by the need to increase the volume of textual data for IR: the previous studies of change-sets, discussed in detail in Chapter 2, show that the majority of change-set descriptions contain less than 15 words and this number, according to Dyer et al. (2013), is less than an average sentence in English language. Hence, for a more verbose description of a source code entity, several change-sets would seem to need to be aggregated. So for example, though Chen et al. (2001) do not justify their decision to include all the change-sets for each line of code that have ever modified that line, it is likely that they tried to expand the volume of textual information by including as many change-sets as were available for that line. In later works by Canfora and Cerulo (2006) and by Zanjani et al. (2014), the aggregation of all change-sets was continued, even though, in these works other, more coarse, levels of granularity, such as file and method, were used. No attempt to aggregate some subset of change-sets (e.g. recent only) was given and the aggregation of all change-sets was used as a “de facto” standard approach.

In this work, the impact of the aggregation of all change-sets was compared to the aggregation of just the most recent change-sets (and aggregation of recent change-sets by a change request), for source code entities. It was found, that for method level source code entities, the aggregation of recent change-sets yields significantly more effective FL (ACIR) than when all change-sets are aggregated. This is surprising given that a method will be touched by more change-sets when all change-sets (as opposed to recent change-sets) are considered and thus will be represented by a larger bag of words. For example, in Rhino, method level entities are described, on average, by 2.11 recent change-sets and by 3.56 change-sets that ever modified the entity (see Table 4.1). This is almost a 50% increase in the number of change-sets and it is likely to translate into a similar increase in the number of words. However, even though, the volume of textual data is decreasing when only recent change-sets are used, the effectiveness of ACIR...
increases. Obviously, it is not simply a case of “less is more”, because, it is not just the smaller number of arbitrary picked change-sets that’s causing the increased effectiveness. It is the aggregation of recent change-sets, that correlates with decreasing numbers of change-sets, that causes the effectiveness increase. Hence, the question is ‘Why do recent change-sets, though smaller in numbers, provide more value in terms of increased effectiveness?’

One possibility is that both functionality and vocabulary of source code entity (as used in change-set description) is evolving over the time. Hence, the textual description of an entity in its most recent state, formed by the aggregation of all the change-sets descriptions, may not accurately describe the current functionality of the entity any more: older change-set descriptions introduce noise, because they are not relevant to the current version of the entity and cause decrease in FL effectiveness.

Additional reasons for the vocabulary of change-set descriptions to change over time include:

1. Older concepts being replaced by new ones while retaining their meaning (i.e. synonyms), as part of re-factoring.

2. New concepts (and hence their descriptions) entering the vocabulary due to the natural evolution of software (Bennett and Rajlich, 2000).

To add more credibility to the first hypothesis above, a recent study suggests that 39% of observed developers engaged in identifier renaming (including using synonyms for renaming) in source code on a constant basis (Arnaoudova et al., 2014). This could also mean that they use new (renamed) concepts in change-set descriptions. As for the second explanation, common code changes such as adaptive and perfective software maintenance seem to be a natural way for the new terms to enter the lexicons of software systems.

The above assumptions, if correct, raise many interesting open questions such as: ‘How exactly vocabularies of change-set descriptions (and software systems in general) change over the time?’; ‘How vocabulary changes correlate with the evolution of software system?’; ‘What is the prevalent type of vocabulary changes (e.g. synonyms or new terms)?’; ‘Does the type of software project (open source
or industry) affect the vocabulary changes?’. Answering these questions could further improve the ACIR approach by allowing the aggregation of change-sets in more sophisticated ways than just taking the most recent change-sets for every entity.

Finally, in this work, the aggregation of recent change-sets by a change request was studied. However, it didn’t yield any significant improvements and, in fact, slightly decreased performance. The slightly worse results for ACIR<sub>RC</sub> could be explained by the inclusion of occasional irrelevant change-set descriptions. This could happen because of the heuristic that was used to map from change requests to change-sets (e.g. a branch may not be associated with just one change, likewise close changes by one author may have more than one purpose).

An interesting question is why inclusion of other relevant change request change-set descriptions does not outweigh these irrelevant inclusions in terms of increased effectiveness. One possible explanation is that the descriptions of change request change-sets employ very similar vocabulary. Hence, the change-set that was the last to touch the code and any change-sets of the same change request before that provide similar textual descriptions. Another explanation is that there are not that many change-sets in a change request that repeatedly touch the same code. Thus, in the future work, other aggregation strategies could be tried. For example, aggregation of recent change-sets by software release.

5.4.3 Exclusion of Management Change-sets from Aggregation

To the best of the author’s knowledge, exclusion of management change-sets has not been studied as part of using change-sets for FL. In this work, the filtering of “management” change-sets was studied and the comparison of ACIR<sub>R</sub> vs ACIR<sub>RF</sub> showed that such filtering, in fact, only decreases the effectiveness. The filtering was based on a naive text classification approach, discussed in Section 3.3.3.5. This approach uses the list of words, identified by Hattori and Lanza (2008) (see Section 2.5.1.3), to exclude the change-sets, that contain those words. The decreased effectiveness was surprising, as the opposite effect was expected.
5.4 Discussion of the Scaled-Up Study Results

A possible explanation is that the words used to filter in the filtering step are pervasive even in change-sets that are not generic management change-sets and therefore causing ACIRRF to mistakenly remove proper change-sets. A review of the list in Section 3.3.3.5 shows how this might be so. For example, there are also going to be change-sets that “upgrade” the handling of medical cards or to include new tax-free allowance thresholds in line with the new government “release” on the topic (a feature).

5.4.4 Comparison of ACIRR vs TFLT

There is a limited data on how FL approaches that leverage change-set descriptions to annotate source code entities compare to textual source code FLT’s:

- Chen et al. (2001) compared their approach to grep and reported that their approach was better (though marginally). However, the set-up of their experiment (unknown queries, non-standard metrics, five subject systems that come from the same KDE package, see Section 2.5.2) raises the possibility that their findings were limited.

- Canfora and Cerulo (2006) evaluated their approach in terms of precision and recall using three subject systems, but did not compare it to baseline/state-of-the-art approaches.

- Zamani et al. (2014) compared their approach to a baseline source code textual FLT and reported the increased effectiveness of their approach. However, their approach was a combination of source code, change-set descriptions, change requests’ descriptions, and interactions (see Section 2.5.2). Moreover, the approach was evaluated using only one subject system.

The results of the current study, where ACIRR was compared to TFLT came as surprising. A pronounced dichotomy was observed in terms of results: half of the subject systems responded better to ACIRR and the other half to TFLT. This finding is analysed in the next chapter.
5.5 Threats to Validity of the Scaled-Up Study

This section presents threats to validity for the scaled-up study. For the list of common threats to validity, that apply to across experiments in this work, see Section 7.3. The threats to validity are organized into construct, internal, external, and conclusion (if and when applicable) as per Wohlin et al. (2000).

There is one notable threat to the internal validity that is specific to the scaled-up study. When aggregating change-sets into change requests, a hard-coded value of 10 days was used for distinguishing change-sets from different change requests (the period between change-sets that are likely to belong to the same change request). This may have impacted on the quality of aggregation. No reliable literature was found on how often developers commit when working on a specific change request. Thus, direct observations of the change-set histories of the selected systems was undertaken for a sample of change requests for each system. These observations suggested that in many cases these are short periods between change-sets belonging to the same change requests, typically of between one and eight days. As a result of these observations, a 10 day upper limit was selected for associating change-sets to the same change request in this study, but it is acknowledged that this is a somewhat arbitrary choice which should be investigated further. No counter-measures were taken to address this threat going forward, because it did not affect the subsequent experiment (the post-hoc experiment, see Chapter 6).

5.6 Application of ACIR: the Effort Data

All experiments (pilot and scaled-up) in this chapter were conducted on a single computer with 2.53GHZ CPU, 6GB RAM, and 128GB SSD. The effort required for each of the steps described in Section 4.7 is presented below:

- Obtaining and configuring the technique. Obtaining the implementation and configuring the technique are once-off/infrequent activities and are accomplished in a similar time for each software system. The effort required
5.6 Application of ACIR: the Effort Data

Table 5.11: The average times of initial search corpus creation.

<table>
<thead>
<tr>
<th>Technique</th>
<th>Average execution times (h - hour, m - minute, s - second)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Method level</td>
</tr>
<tr>
<td>ACIR_A</td>
<td>1h24m28s</td>
</tr>
<tr>
<td>ACIR_R</td>
<td>7m37s</td>
</tr>
<tr>
<td>ACIR_RC</td>
<td>2h16m36s</td>
</tr>
<tr>
<td>ACIR_RF</td>
<td>7m49s</td>
</tr>
<tr>
<td>TFLT_B</td>
<td>10m27s</td>
</tr>
</tbody>
</table>

to accomplish these activities depends on the expertise of the user (developer). Though it is hard to estimate the effort required for this activity, a duration of up to a half an hour seems reasonable for a novice user.

- Creating a search corpus. This activity is infrequent: it has to be performed only when the search corpus is initially created or when the search corpus is updated to reflect code changes. The execution time of this activity was measured in milliseconds, starting just before the launch of a technique and ending just after the corpus was created. Such a measurement method can be inaccurate because of interference of other processes that run on a computer (that is, other processes can consume the CPU and RAM and hence delay the execution of a technique). The average time (across the 8 subject systems of the scaled-up experiment) of the initial search corpus creation is shown in Table 5.11. There are three noticeable outliers with respect to the execution times: method level ACIR_A, method level ACIR_RC, and file level TFLT_B. The longer execution times of the two former configurations (ACIR_A and ACIR_RC) are likely due to more complicated historical analysis involved at a finer level of granularity (see Section 3.5). The file method TFLT_B is the fastest because no parsing is required and files can be passed directly (after preprocessing, see Section 3.4) to the IR engine. The cost to update a search corpus should be a fraction of the initial execution times (because a smaller sub-set of source code has to be updated). It should be
noted that the time required for the best configuration of ACIR (ACIR\textsubscript{R} at method level) is on average just 7.5 minutes approximately.

- Querying a search corpus. This is a frequent activity that a user (developer) would use in his/her daily work. For all systems/configurations querying times never exceeded a few seconds for a single query and in general completed within one second on average.
Analysing the Post-hoc Findings: System Characterization and Hybrid Approach

The results of comparing ACIR to TFLT_B revealed an interesting dichotomy of subject systems: half of them responded better to ACIR_R and the other half responded better to TFLT_B.

In this chapter, first, the characteristics of the subject systems are studied to determine why this might be so. Given that the change-set based approach (ACIR) provided very different candidates than the source-code based approach (TFLT_B), the impact of merging the two approaches was then assessed. The resultant hybrid approach incorporates both change-set descriptions and source code to describe source code entities. The methodology and research questions of a post-hoc study that compares this hybrid approach to standalone ACIR_R and TFLT_B are then presented and this is followed by the results and discussion.

6.1 System Characterization

The dichotomy, discovered in the previous chapter when comparing ACIR and TFLT_B, was notable and demanded further probing: it was reasonable to assume that systems of Group A (see Section 5.3.5) might share some common
6.1 System Characterization

Table 6.1: Similar features of subject systems

<table>
<thead>
<tr>
<th>Subject</th>
<th>Comments % of LOC</th>
<th>First Commit Year</th>
<th># Change-sets</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Group A</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rhino</td>
<td>22</td>
<td>1999</td>
<td>3372</td>
</tr>
<tr>
<td>Mylyn.Tasks</td>
<td>22</td>
<td>2005</td>
<td>8667</td>
</tr>
<tr>
<td>JGit</td>
<td>64</td>
<td>2009</td>
<td>4425</td>
</tr>
<tr>
<td>Jetty</td>
<td>31.7</td>
<td>2009</td>
<td>12215</td>
</tr>
<tr>
<td><strong>Group B</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ant</td>
<td>75.59</td>
<td>2000</td>
<td>13231</td>
</tr>
<tr>
<td>Hudson</td>
<td>59</td>
<td>2011</td>
<td>1555</td>
</tr>
<tr>
<td>JMeter</td>
<td>53</td>
<td>1998</td>
<td>12136</td>
</tr>
<tr>
<td>Eclipse.</td>
<td>55</td>
<td>2001</td>
<td>6273</td>
</tr>
<tr>
<td>Platform.Text</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

features that are different from those in Group B. A number of characteristics was compared that may have an impact on the results:

- Lines of code;
- Average number of words that are used to describe a source code entity;
- Number of bugs in these systems;
- Number of change-sets;
- Number of files and methods;
- Amount of comments in code;
- System’s age.
6.2 Hybrid Textual Source Code and Meta-Data Approach: BACIR

The only characteristic that was found to be different between Group A and Group B systems was the number of comments in source code (calculated as the proportion of comments to lines of code) (see Table 6.1). It seems that Group A systems tend to have fewer source code comments than systems of Group B: 34.92% on average in Group A and 60.65% on average in Group B. Another, less significant characteristic, that could potentially differentiate these two groups is the recentness of systems’ deployment (shown as the year of first deployment) (see Table 6.1). In general, the systems of Group A seem to be more recent, than the systems of Group B on average: 11.5 years have passed, on average, (calculated by subtracting the first commit year from the year 2017) since the initial commit in Group A, compared to 14.5 years, on average, since the initial commit in Group B. However, it should be noted that Rhino in Group A is the second oldest system in the cohort and Hudson in Group B is the newest system in the cohort. This, coupled with the small number of systems studied, makes this trend highly provisional.

Interestingly, the sheer number of change-sets does not seem to interact with any of these groups: both groups have similar numbers of change-sets. (Actually, because Group A have a better response to ACIR one would expect that the systems of this group would have more change-sets, but they have less: 7170 on average for Group A versus 8299 on average for Group B.) Due to the small sample size these provisional findings cannot be tested for statistical significance.

6.2 Hybrid Textual Source Code and Meta-Data Approach: BACIR

Given the observed dichotomy in the previous section, it would be interesting to probe how efficient the hybridization of source code and change-set textual sources is as an IR approach? In this section, such a hybrid approach, called BACIR (Baseline and ACIR), is presented and assessed.
6.2 Hybrid Textual Source Code and Meta-Data Approach: BACIR

6.2.1 The Two Data Sources: Source Code and Change-set Descriptions

Change-set descriptions are textual messages, written by a developer when he/she is committing their work to a VCS.

The code of the software system is another data source that is the de facto standard in TFLTs (Marcus and Haiduc, 2013). Source code is written in one of the programming languages.

Still, there is meaningful semantic information that could be discovered in source code and that could assist FL. Usually this information comes from comments, identifiers, and literals (Dit et al., 2013a).

6.2.2 BACIR: Combining Source Code and Change-set Sources

To study the impact of combined source code and change-set lexicons, the TFLT_B and ACIR approaches had to be merged into one technique: BACIR. As shown in Figure 6.1, source code and the VCS both provide the input data for BACIR approach. The approach works in several steps:

- Partition the source code into a set of source code entities of certain granularity as discussed previously in Section 3.2 and Section 3.4 (the resultant set of source code entities is used in the next steps).

- Annotate the source code entities with source code lexicons and change-set descriptions:
  - For the ACIR part, change-sets are extracted from VCSs and matched against the source code entities. The text preprocessing for change-set descriptions is used, as described in Section 3.3.3.6. Also, for the ACIR part the approach focuses on one specific configuration of ACIR, the non-filtered method level approach including the most recent changesets only, ACIR_R. This configuration was found to be the most efficient according to the experiment presented in Chapter 5. Henceforth, the
BACIR\textsubscript{R} notation is used to specify the particular configuration of ACIR, used in BACIR.

- For the TFLT\textsubscript{B} part, textual data is parsed from source code for annotation. The text preprocessing for source code is similar to that used for change-sets, but, additionally, programming language keywords were removed from source code.

- After the previous step, each textual document, representing source code entities is a bag-of-words (where the words are mixed, coming from both change-set descriptions and source code). The documents are indexed using the IR engine (VSM model).

The implementation of BACIR\textsubscript{R} uses the same third-party libraries as discussed in Chapter\textsuperscript{3}. It is also a part of FUSIX library (see Chapter\textsuperscript{3}).

### 6.3 Empirical Methodology

In this post-hoc study the research question and hypothesis are defined as shown below:

**RQ1:** How efficient is the hybridization of source code and change-set textual sources in the IR approach, BACIR\textsubscript{R}?
6.4 Results: BACIR$_R$ vs ACIR$_R$ and TFLT$_B$

**Table 6.2:** MAP for ACIR$_R$, TFLT$_B$, and BACIR$_R$

<table>
<thead>
<tr>
<th>Subject system</th>
<th>Rhino</th>
<th>Mylyn.Tasks</th>
<th>JGit</th>
<th>Jetty</th>
<th>Ant</th>
<th>Hudson</th>
<th>Jmeter</th>
<th>Eclipse. Platform.Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACIR$_R$</td>
<td>10.48</td>
<td>40.08</td>
<td>35.77</td>
<td><strong>54.05</strong></td>
<td>4.59</td>
<td>4.85</td>
<td>8.33</td>
<td>7.98</td>
</tr>
<tr>
<td>BACIR$_R$</td>
<td><strong>12.03</strong></td>
<td><strong>41.81</strong></td>
<td><strong>38.29</strong></td>
<td>52.71</td>
<td><strong>13.02</strong></td>
<td><strong>10.74</strong></td>
<td><strong>15.72</strong></td>
<td><strong>10.88</strong></td>
</tr>
</tbody>
</table>

H$_{01}$: The combination of change-set annotation approach and baseline approach does not result in a significant improvement over any of these standalone approaches.

An identical methodology, as defined in Chapter 4, was used to answer the research question excepting that the techniques employed were ACIR$_R$, TFLT$_B$, and BACIR$_R$. Particularly: the eight subject systems were re-used, the MAP metric was used to assess and compare the results, and the gold sets were re-used. The effort was not measured in this experiment, because the techniques were compared at method level granularity, the granularity that was found to be the most efficient for ACIR.

6.4 Results: BACIR$_R$ vs ACIR$_R$ and TFLT$_B$

The results of experiment data are detailed in this section. Overall BACIR$_R$ outperforms the other configurations in seven out of the eight systems and the difference in the 8th system is negligible (see Table 6.2).

First, a multi-way comparison of TFLT$_B$, ACIR$_R$, and BACIR$_R$ was performed, using Friedman test. To adhere to this test the data has to meet certain criteria:

- There has to be one set of samples, measured under at least three conditions. In this case the sample set consists of subject systems that are measured when three approaches are applied: ACIR$_R$, TFLT$_B$, and BACIR$_R$. 

156
6.4 Results: BACIR\textsubscript{R} vs ACIR\textsubscript{R} and TFLT\textsubscript{B}

Table 6.3: Wilcoxon signed rank statistical test of TFLT\textsubscript{B}, ACIR\textsubscript{R}, and BACIR\textsubscript{R} MAP distributions

<table>
<thead>
<tr>
<th>Approach</th>
<th>N</th>
<th>Mean</th>
<th>StDev</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>TFLT\textsubscript{B}</td>
<td>8</td>
<td>12.5413</td>
<td>3.58911</td>
<td>8.47</td>
<td>18.41</td>
</tr>
<tr>
<td>ACIR\textsubscript{R}</td>
<td>8</td>
<td>20.7663</td>
<td>19.43816</td>
<td>4.59</td>
<td>54.05</td>
</tr>
<tr>
<td>BACIR\textsubscript{R}</td>
<td>8</td>
<td>24.4000</td>
<td>17.00733</td>
<td>10.74</td>
<td>52.71</td>
</tr>
</tbody>
</table>

Friedman rank sum test

<table>
<thead>
<tr>
<th>Comparison</th>
<th>Chi-squared</th>
<th>p-value</th>
<th>Significant</th>
</tr>
</thead>
<tbody>
<tr>
<td>TFLT\textsubscript{B} vs ACIR\textsubscript{R} vs BACIR\textsubscript{R}</td>
<td>9.25</td>
<td>0.009804</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 6.4: Pairwise comparisons using Wilcoxon signed rank test

<table>
<thead>
<tr>
<th></th>
<th>TFLT\textsubscript{B}</th>
<th>ACIR\textsubscript{R}</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACIR\textsubscript{R}</td>
<td>0.641</td>
<td>-</td>
</tr>
<tr>
<td>BACIR\textsubscript{R}</td>
<td><strong>0.023</strong></td>
<td><strong>0.031</strong></td>
</tr>
</tbody>
</table>

MAP improvement %

<table>
<thead>
<tr>
<th></th>
<th>BACIR\textsubscript{R}</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>+95%</td>
</tr>
<tr>
<td></td>
<td>+17%</td>
</tr>
</tbody>
</table>

- The samples have to be randomly selected. The subject systems were selected randomly, as described in Section 4.4.

- The samples have to be distributed non-normally: Shapiro-Wilk’s test showed p-values below 2.2e-16 for data generated by ACIR\textsubscript{R}, TFLT\textsubscript{B}, and BACIR\textsubscript{R} (see Figure 6.2 for distribution).

The Friedman test of MAP values shows strong evidence that these values come from different distributions (see Table 6.3). Looking at Table 6.3 it is noticeable that the mean MAP values of BACIR\textsubscript{R} are better than those of TFLT\textsubscript{B} and ACIR\textsubscript{R}.
6.4 Results: BACIR\textsubscript{R} vs ACIR\textsubscript{R} and TFLT\textsubscript{B}

![Figure 6.2: Distribution of average precision data: x-scale - average precision, y-scale - density](image)

Pairwise comparison of these three approaches using Wilcoxon signed rank test proves that there is a significant difference between ACIR\textsubscript{R} and BACIR\textsubscript{R}, and TFLT\textsubscript{B} and BACIR\textsubscript{R} (see Table 6.4). BACIR\textsubscript{R} improves upon ACIR\textsubscript{R} by 17% and upon TFLT\textsubscript{B} by 95% in terms of MAP. ACIR\textsubscript{R} and TFLT\textsubscript{B} do not show significant difference to each other, due to the reasons discussed earlier in Section 5.3.5. Therefore, the combination of approaches, BACIR\textsubscript{R}, is more efficient for FL than either standalone TFLT\textsubscript{B} or ACIR\textsubscript{R} approaches. Hence, H\textsubscript{01} could be safely rejected.

6.4.1 Application of BACIR: the Effort Data

The BACIR technique was studied in the same hardware setting as described in Section 5.6. The effort required to set up and configure BACIR is equivalent to that described for ACIR in Section 5.6. The execution times constructing the search corpus are shown in Table 6.5. As can be seen from Table 6.5, the execution times at method level are similar for ACIR\textsubscript{R}, TFLT\textsubscript{B} and BACIR\textsubscript{R} approaches. Finally, the querying times are similar to those described in Section 5.6.
6.5 Discussion: Systems’ Characterization and BACIR$_R$

Table 6.5: The average times of initial search corpus creation (method level).

<table>
<thead>
<tr>
<th>Technique</th>
<th>Average execution times (h - hour, m - minute, s - second)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACIR$_R$</td>
<td>7m37s</td>
</tr>
<tr>
<td>TFLT$_B$</td>
<td>10m27s</td>
</tr>
<tr>
<td>BACIR$_R$</td>
<td>7m43s</td>
</tr>
</tbody>
</table>

6.5 Discussion: Systems’ Characterization and BACIR$_R$

In this section a discussion of systems’ characterization and the analysis of results, comparing BACIR$_R$ to ACIR$_R$ and TFLT$_B$, are presented.

6.5.1 The Impact of System Characteristics on the Use of ACIR

In the pilot study of this work, ACIR was initially compared to existing TFLT$_s$. These initial observations suggested that the effectiveness of ACIR might be at least as good as that of existing TFLT$_s$. Inspired by these initial findings, the study was extended to include eight subject systems and compared against the implementation of the baseline state-of-the-art TFLT$_B$. The results of the latter experiment uncovered a strongly pronounced dichotomy with regard to the eight subject systems. The ACIR$_R$ approach outperformed TFLT$_B$ when applied to half of the subject systems: Rhino, Mylyn, JGit, and Jetty. Whereas the TFLT$_B$ outperformed ACIR when applied to the other half of the subject systems: Ant, Hudson, Jmeter, and Eclipse. It was hypothesized that some characteristics of the subject systems are responsible for the dichotomy that was observed.

Several metrics of the subject systems were inspected (see Section 6.1). However, the only metric that seemed to be relevant was the ratio of source code comments to lines of code (see Table 6.1). Essentially, the software systems of the first group A (those that benefit when ACIR is applied) tend to have a lesser ratio of source code comments than the systems of the second group B, which
6.5 Discussion: Systems’ Characterization and BACIR

have a higher ratio of comments. The systems, where the ratio of comments in the source code is low, seem to benefit when an alternative data source such as change-sets descriptions is applied. It seems that in some software systems (Group B) vocabulary is more plentiful because comments are plentiful, and so, the annotation by change-sets is less beneficial.

Another less pronounced metric that could be considered is the seeming correlation between the date of inception and the dichotomy of subject systems. The systems of Group A tend to be younger overall than the systems of Group B, which tend to be older. Given the small cohort, and the outliers within this small cohort, this finding is highly provisional. However, if true, one possible explanation is that older software systems were started at a time when documentation outside the source code was less emphasized. Hence, the code documentation policies in such software systems (Group B) historically put more emphasis on source code documentation than other documentation, including that of VCSs. A similar explanation might be less formalized use of VCS descriptions in older systems. Alternatively, it might simply be that older systems (Group B) have built up more comments. Ultimately, these are just post-hoc hypotheses that would need further empirical evaluation.

Overall, the most important takeaway of this finding is that system’s characteristics (as possibly measured by metrics) may play an important role in improving the selection of appropriate FLTs. Software metrics were applied in many software engineering fields such as cost estimation, software project management, quality and security evaluation, and software maturity assessment [Fenton and Bieman, 2014]. However, in FL no work or consideration has been given to selecting the appropriate FLT based on system characteristics/metrics. Indeed, this might even be applicable with respect to the configuration of the FLT chosen [Dit et al., 2013a; Marcus and Haiduc, 2013]. It has been suggested in this study, that the metrics characterizing software systems could have a profound effect on the effectiveness of FLT. Moreover, this finding is not confined to just TFLTs. For example, measuring the number of available test cases and their quality, could suggest if a dynamic FLT should be used. This lesson is only beginning to be implicitly learned by the FLT community: for example, more recently, a genetic algorithm has been used to pick the best-performing configuration of LDA for FL
6.5 Discussion: Systems’ Characterization and BACIR\(_R\)

(Panichella et al., 2013; Sun et al., 2015). It seems that the further research in FL could see an emergence of profile-based FLTs, where the profile of a software system created from its metrics is used to select and configure the appropriate FLT.

6.5.2 Hybridization of Source Code and Meta-data TFLTs: BACIR\(_R\)

Combinations of various approaches is an actively studied topic in FL. In 52\% of FLTs’ publications, identified by Dit et al. (2013a), the combination of at least two different approaches to FL was studied. In the literature review of more recent articles in this field conducted as part of this work an even higher ratio of such publications was identified, 59\%, suggesting an increasing trend, going forward. According to the classification of FLTs used in this work, combinations of FLTs usually combine dynamic, static structural, and static textual (including both source code and meta-data approaches) approaches. The latter, particularly hybrid approaches that employ VCS meta-data, were not thoroughly studied and contributed the least number of hybrid FLTs. This work is the first to study the potential for hybridization of source code and VCS meta-data: it does it in isolation, with a large number of distinct subject systems, providing statistical significance testing, and the hybridization is performed using the best-performing configuration of the ACIR approach identified in advance.

In this study, the best-performing configuration of ACIR (ACIR\(_R\)), was combined with the baseline TFLT\(_B\) approach. The combination of these approaches, BACIR\(_R\), was applied to eight subject systems previously studied in this work and the assessment followed the methodology described earlier. The results generated by BACIR\(_R\), showed that BACIR\(_R\) outperforms both standalone ACIR\(_R\) and TFLT\(_B\). The immediate conclusion suggests that it is efficient to combine the two data sources (source code and change-set descriptions). It also suggests that these two data sources are both meaningful for FL and distinct. Indeed, if any of these two conditions (meaningful and distinct) were not true, the effectiveness of BACIR\(_R\) would not improve significantly over the standalone approaches of ACIR\(_R\) and TFLT\(_B\).
These findings raise a question: ‘How different are these data sources?’ It is possible that change-set descriptions are more likely to contain the words that are more abstract and closer to what could be a feature description. (Certainly the comparison of ACIRr to TFLT suggests that in some cases, but not all.) Indeed, change-sets touch many places in source code, hence they need to use the terms that are broad and abstract enough to be able to apply to all these pieces of code, although the down-side of that is that their ability to discriminate between different parts of the system is lessened. Source code terms in comments and identifiers are, presumably, more narrow/specialized and, hence, less likely to match with the broader/more abstract feature description terms, but then “specialist words” are good too. This could explain why the improvement in effectiveness was more pronounced for TFLT_{B}, where 95% improvement was recorded, and less so for ACIR_{R} with 17% improvement.

6.6 Summarizing the Results of Experiments

To summarize this chapter:

- Software characteristics (as possibly measured by metrics) can impact on the selection of an appropriate FLT.
- Hybridization of ACIR and TFLT_{B} is more effective than using these standalone approaches.
Conclusions and Further Research Directions

In this work a new way (ACIR) of associating lexicons (derived from VCSs’ change-sets) with source code entities was assessed for textual FL. The optimal configuration for such an approach was characterized and the approach was compared to a state-of-the-art IR technique VSM(TFLT_B). Finally, the combination of TFLT_B and ACIR was assessed.

The optimal configuration for ACIR was empirically identified as being applied at method level, utilizing the most recent change-sets only and applying no filtering to the sets of “management” change-sets, although a more sophisticated means of determining management change-sets may serve to reverse the last guideline.

Probably the most encompassing finding in this study is the need to characterize proposed FLTs in general to identify an optimal configuration for these FLTs. Here the differences between different configurations of ACIR were significant, and not always in a direction that was intuitive. Indeed, the results also suggest that systems under study might also be usefully characterized in advance to help select the appropriate FLT. In the past configuration of FLTs and characterization of subject systems have not been considered as first-class entities in research papers, and this work suggests they should be, going forward.

This chapter proceeds in the following order: the contributions of this work are enumerated; the answers to the original research questions are given; the
threats to validity of this work are discussed; the implications for researchers and practitioners are presented; and the chapter is concluded with discussion of the future work.

7.1 Contributions

The original contribution of this work to the existing knowledge includes:

- A novel textual FLT (ACIR), targeted at the issue of vocabulary paucity in source code, which was proposed and implemented. It is based on annotating the code with change-set descriptions. This technique is optimized for:
  - Granularity of the targeted source code entities;
  - Recentness of change-sets;
  - Filtering of management change-sets.

- The determination that selection of FLT s should be based, in part, on system characteristics and enumerating some of these system characteristics for ACIR.

- Evaluation of that approach showing that it is competitive against a state-of-the-art TFLT.

- A hybrid approach (BACIR) using the source code and change-sets for annotations that shows significant improvement on each constituent textual FLT component (ACIR\textsubscript{R} and TFLT\textsubscript{B}).

7.2 Answering the Research Questions

RQ1: How can change-sets be leveraged to improve TFLT\textsubscript{IR}s?

One answer to this question is embodied in the ACIR approach, that was presented in Chapter 3. The ACIR approach allows for aggregation of change-set descriptions to annotate source code entities (using change-set
descriptions that collectively describe source code entities) for FL. Combined with IR, it allows for indexing and searching for source code entities, given a search query. The input to ACIR is source code and the associated VCS, that stores a set of change-sets. ACIR relies on the descriptions of change-sets to generate the textual content of documents for IR, where each document is representing a source code entity.

**RQ2: What is the best configuration towards such improvement of TFLT$_{IR}$s in terms of change-sets recentness, type, and granularity?**

This research question is answered by the empirical study in Chapter 5. Change-set descriptions/annotations could be aggregated in several distinct ways to address the vocabulary-paucity problem. Particularly, the granularity, the age of change-sets, and filtering of change-sets were assessed as potential properties that could affect the effectiveness of annotation by change-sets.

The first finding to address this research question is that applying the method level of granularity was found to be more efficient for ACIR than the file level of granularity, when locating entities overall. This is in line with previous research on the effects of granularity for TFLTs (Tantithamthavorn et al., 2014). It seems that TFLT, in general, and ACIR, in specific, are better used at method level granularity.

Another interesting finding of this work suggests that inclusion of all available change-sets has negative impact on the efficiency of ACIR at method level granularity. A strong, positive, and statistically significant correlation between recentness of change-sets and the MAP results was observed for all subject systems. Quantitatively, ACIR$_R$ outperforms ACIR$_A$ by 178% in terms of MAP. This possibly reflects the likelihood that more recent VCS content, is more aligned with the current version of the source code.

Interestingly, the aggregation of recent change-sets by change request showed worse effectiveness than just the most recent (the last to touch code) change-sets. The ACIR$_R$ approach was better than ACIR$_{RC}$ by 5.79% on average.
and this difference was significant at the 0.05 level of significance. Such results, on the one hand, could be attributed to some number of false positive change-sets that were included when aggregating them by change request, based on the heuristics used or, on the other hand, to homogeneity of change-set descriptions in a change request.

Surprisingly, filtering of management change-sets, using text classification based on generic management keywords showed statistically significant decreases in performance. In fact, ACIR$_R$ was 12% better in terms of MAP than ACIR$_RF$. Such behaviour could be attributed to the reductionist technique applied, possibly removing function-oriented change-sets that contain extemporaneous management keywords. Future work could investigate if other more sophisticated techniques of classification would improve on this.

By systematically comparing and eliminating the configuration that performed the worst, it was found that the answer to RQ2 is that the non-filtered method level ACIR configuration that includes the most recent change-sets shows the best results. This configuration was found to be significantly more efficient than others.

**RQ3: How does the best-practice configuration of the new approach (ACIR), as defined by the answer to the second research question, compare to a state-of-the-art baseline TFLT$_B$ approach that leverages meaningful lexicons in source code?**

In RQ2, the best-performing configuration of ACIR, ACIR$_R$, was determined and this configuration was compared with a baseline TFLT$_B$ approach. A very pronounced dichotomy of subject systems was found, notable when TFLT$_B$ and ACIR$_R$ are applied to them. It was found that the ACIR$_R$ technique performed better for four of the eight systems (Group A), whereas the TFLT$_B$ was found to be better for the other four systems (Group B). So, in answer to RQ3: mixed results were observed where ACIR worked better for one half of the software systems studied and less for the other half.
RQ4: What system characteristics respond well to ACIR as opposed to TFLT_B?

It is reasonable to assume, that systems of type Group A might share some common features that differ to Group B. Several metrics of the subject systems were inspected and one of them seem to be relevant: the ratio of source code comments to lines of code. Systems that have less comments in source code responded better to ACIR. Another less pronounced characteristic that could be considered is the age of the software system. Systems that are younger seemed to respond better to ACIR. These provisional findings, however, need further empirical evaluation.

RQ5: How efficient is the hybridization of source code and change-set textual sources in an IR approach?

Given the diverse nature of systems found in RQ3 and RQ4 the combination of ACIR and TFLT_B was assessed. It was found that a combination of both techniques, BACIR_R, significantly outperforms both ACIR_R (by 17%) and TFLT_B (by 95%): data fusion of change-sets and source code lexicons leads towards a positive increase in FL performance.

Thus, the answer to RQ5 is that it seems that the combination of ACIR_R and TFLT_B is significantly more efficient.

To sum up and to answer the main research question “Can change-set annotation of code assist FL?”:

- The proposed approach, that leverages change-set descriptions for source code annotation (ACIR), is effective.

- There are several ways of aggregating change-sets and the ACIR_R, at method level, non-filtered, and including only the most recent change-sets is better at assisting FL.

- This configuration seems to work better for FL in systems that have less source code comments.

- In combination with TFLT_B, ACIR generates more efficient results than standalone TFLT_B and ACIR_R approaches.
7.3 Threats to Validity

In this section, the factors that might impact on the validity of the experiments of this work are presented. The study-specific threats are presented in their corresponding sections: Section 4.2.2 for the pilot study and Section 5.5 for the scaled-up study. Here only common threats to validity are presented. The threats to validity are organized into construct, internal, external, and conclusion (where applicable) validity as per Wohlin et al. (2000).

7.3.1 Construct Validity

As discussed in Section 4.5, the most recent successive change requests were picked to construct a test sample. This approach might cause several problems, because such a sample is less randomized. For instance, when selecting successive change requests, one might accidentally test a series of modifications to some particular part of the code (in essence, concentrating on a sub-set of features).

Also, the document corpus was not re-indexed for each of these change requests individually to include the changes of each preceding change request in a sample. Instead, the corpus was created at a threshold (change-set) that would satisfy all of these change requests. Therefore, those change requests in a sample which are closer in time to this common threshold may have more correct results than those change requests further away in time. This may have impacted on the findings for ACIR, but would probably have impacted on them negatively, if at all. Consequently this validity issue, if impactful would have only made ACIR appear less effective.

The re-enactment process used also leaves the possibility that change-sets not specifically related to functionality (and thus not applicable to FL) are used to annotate source code.

An alternative would be to conduct a controlled experiment with human participants. Such an experiment can possibly allow for more rigorous quantitative and qualitative assessment of the effectiveness of the techniques used in this work and the applicability of the techniques in settings that are closer to real world scenarios. However, the re-enactment approach is standard in the field because
it allows for larger scale experimentation, in this case with 600 sample change requests. Human-based experimentation would prohibit this size of study.

### 7.3.2 External Validity

When selecting subject systems for this experiment, several acceptance criteria were applied, such as LOC, the number of change-sets and the number of files (see Section 4.4). With respect to external validity, the systems should be as diverse as possible, so that the results are generalizable to all systems. Yet, there are several threats related to the representativeness of these subject systems:

- The subject systems of this study come from medium size software systems. The exclusion of small systems should not pose a validity threat, since FL in such systems is less relevant. On the other end of the spectrum, large systems are more of a concern and should be addressed in future work.

- Also, system-selection criteria, described in this work, restricts the variability of systems. For example, all the subject systems are written in Java. This might have an impact on the amount of meaningful textual data in source code. Java is known to be verbose: identifiers and Javadoc comments are usually self-descriptive. This might not be the case in older languages like FORTRAN77, for example, which have limitations on the size of their identifiers.

- Likewise, all of the subject systems come from open source systems - no commercial systems are utilized. Change-sets in the open community may be less formalized than in commercial systems.

### 7.3.3 Conclusion Validity

In this work eight subject systems were used. This number of subject systems is larger than those used by the authors of most of the closely related work (see Table 2.6) and is on par with the most recent studies in FL. For example, Scanniello et al. (2015) also used eight subject systems in their work and Zamani et al. (2014) used only 4. Yet, this pool of subject systems should be further
expanded. For example, the dichotomy of subject systems, observed in this work, should be further evaluated on a larger sample of subject systems to identify more statistically significant trends.

Also, as pointed out in Section 4.4 the sampling protocol was limited by the need to have sufficient change-sets to annotate the source code, as a basis for the ACIR approach. Henceforth, evaluation of the technique should only happen on systems with large (over 2000) change requests, where an adequate population of change requests exists to annotate the source code and to provide an appropriate sampling size for evaluation.

7.4 Implications for Researchers and Practitioners

The findings of this study suggest that certain configurations of ACIR result in a significantly better performance than others. If practitioners intend using ACIR, they should use only recent change-sets, at method level granularity and not filter management change-sets naively. More generally, this has a direct implication for researchers, since until now there has been little effort expended in identifying and assessing the optimal configuration for new FLTs. This work suggests that such an assessment may give significant improvements. Currently, the research direction in this field has geared itself towards combinations of approaches and query reformulation. Instead, when proposing novel FLTs researchers should concentrate on identifying the best performing configuration in isolation and documenting it for readers who may hope to adopt the technique in practice or compare their FLT against it. Only with that best-configuration established should they move to using the approach in a hybrid technique.

A finding with large impact for both researchers and practitioners is that the characteristics of software systems can have a significant impact on the effectiveness of the FLT employed. In this study ACIR was found more effective for systems with a lesser proportion of comments (and plausibly for newer systems). This holds out the prospect of recommender systems where automated analysis of software systems, leads to selection of an appropriate FLT. Currently,
7.5 Final Conclusions and Future Work

FLTMs employ “one-approach-fits-all” strategy when locating features in different systems.

7.4.1 Applicability of the Technique to Unknown Software Systems

In this work, it was found that ACIR configuration at method level including the most recent non-filtered change-sets is the most effective configuration when compared to the others. However, for some software systems TFLT_B approach seems to perform better. Thus, when approaching an unknown software system a user (developer) should use preprocessing to identify the relevant characteristics of that system to make an informed decision. It was found that the smaller amount of comments in code and (possibly) a younger age can suggest that ACIR should be used. In addition, ACIR should only be used if a large set of change-sets is available for annotation of the source code.

Then to use the technique, a user would have to follow the guidelines outlined in Section 4.7:

- To obtain the implementation and to configure the technique.
- To create a search corpus for a software system.
- To use the technique by running queries.

Section 5.6 provides a guidance of what could be expected in terms of effort.

7.5 Final Conclusions and Future Work

Over more than a decade of FL research, scientists have generated a fair number of FLTMs: 60 FLT papers were identified by Dit et al. (2013a) and 27 were identified as a part of the literature review in this study. Currently, the FL research focus is directed at FLTMs that employ hybridization of other approaches (there are over half of the papers describing such approaches in the literature). However, such approaches were rarely analysed to find the best-performing configurations.
and were applied to all software systems equally as a typical paradigm of “one-
approach-fits-all”. While the need for more sophisticated tuning of FLTs has
recently been recognized Panichella et al. (2013), this should be a growing trend
in the field going forward. The intention should be to look at existing FLTs to
assess if the configuration of other state-of-the-art techniques in the field impacts
on effectiveness.

Additionally, future work should focus on expanding the grouping and clas-
sification of change-sets. In terms of grouping, methods of aggregation of recent
change-sets by change request could be further investigated along with aggre-
gation of release/version change-sets. In terms of classification, there could be
a more sophisticated mechanism for removing less-impactful change-sets from
data-sets, thus improving the performance of ACIR further.

In this work another interesting finding was identified that suggests a new,
meta-level FLT. It was found that different subject systems respond differently to
source-code and change-sets’ based FL. Therefore, future research should focus on
identifying characteristics of software systems that correlate with certain FLTs.
Indeed, there are other non-source-code characteristics that might suggest the
selection of differing FLTs. For example in dynamic FLTs, the amount of source
code tests (that could be used as scenarios) and the coverage of source code by
these tests could be used as one of the characteristics. Scenarios are essential for
dynamic FLTs (see Section 2.2.2.2).

Alternatively, this meta analysis could be applied to a whole range of differ-
ent characteristics like company organization, code author, and architecture. For
example, if the team is not very stable it suggests that naming conventions will
be less consistent and so textual techniques might have less leverage. In contrast,
a stable development team would suggest more consistent usage of the conven-
tions and textual techniques may be more appropriate. However even within this
structure the individual programming styles of the code authors might further
direct the selection of an appropriate FLT.

Based on the characteristics of a software system, a smart meta-FLT, instead
of treating all software systems equally, could analyse their properties and select
a personalised FL approach and/or data source. Such personalised approaches
are finding their way in other research domains, such as medicine
7.5 Final Conclusions and Future Work

The assumption is that a different set of genes in every human should command a different type of treatment. Similar assumption could be applied to software systems, where software metrics would command the usage of the appropriate FLT. For example, a recommender system in conjunction with genetic algorithms could be used. The genetic algorithms would identify the best-performing configuration for a given software system. The recommender system would learn the characteristics of this system and the “treatment” (the best-performing configuration of some FLT) that was applied. The next time the recommender system encounters similar software system it would have the “treatment” ready.

The initial reading of the FL literature suggests that new FTLS predominate and that thorough evaluation of these approaches is lacking. Specifically the techniques proposed are not always evaluated against standard, baseline techniques, for comparison purposes, and empirical design details required for replication are not presented in the articles. In this work, the ACIR approach was compared to an openly available baseline TFLT_B, and the replication procedures are clearly outlined in Section 4.7. Also, The application itself with data from the empirical studies in this thesis, is accessible at this address [1] All these facilities raise the replicability and comparability of the work presented here.

It is noted that the diverse artefacts (not just source code entities) covered by change-sets mean that the technique proposed in this thesis is not only suitable for FL in code, but also for FL in XML documents and documentation in general, providing a wider, more encompassing target for FL and in areas like architecture recovery.

Features describe the functionality a software system has to provide (based off the requirements) and its software architecture provides a macro-structuring for that software system. It is likely that such architectural descriptions can help guide feature location. As mentioned earlier, for example, features are likely to be partially located in all layers of a layered architectural system. Likewise, it is probably true to say that all user-observable features would be partially located on the client and server sides of a distributed system architecture. In addition, similar features could be implemented in a similar manner, using similar


173
components and localized to similar parts/layers of a software system. This potential relationship between architecture and feature location could lead to the further characterization of subject systems for appropriate FLTs.

Architecture recovery is concerned with the restoration of software architecture to ensure better maintainability and modularity of a software system. Interestingly, recent studies suggest that software architects would like to see a feature-based perspective of their systems (Buckley et al., 2015). This suggests architectural views in line with logical view by Kruchten (1995), where high level functional abstractions are presented to the architect. Given the orthogonal nature of architecture (the how) and features (the what), alluded to above, it is likely that such a view should superimpose functions on architectural abstractions or allow navigation from one perspective to the other with ease.
Appendix A: Listing of All FL Papers Used in Taxonomic Structure in Phase 1

<table>
<thead>
<tr>
<th>FLT papers</th>
<th>Dynamic</th>
<th>Static</th>
<th>Textual</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Structural</td>
<td>Source code</td>
<td>Meta-data</td>
</tr>
<tr>
<td>Papers from Dit et al. and Rubin and Chechik</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wilde1995</td>
<td>+</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wilde1992</td>
<td>+</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Eisenberg2005</td>
<td>+</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wong1999</td>
<td>+</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Eisenbarth2001a</td>
<td>+</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Eisenbarth2001b</td>
<td>+</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Safyallah2006</td>
<td>+</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sartipi2010</td>
<td>+</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Edwards2006</td>
<td>+</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bohnet2008</td>
<td>+</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chen2000</td>
<td>+</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Robillard2002</td>
<td>+</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Robillard2007</td>
<td>+</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Robillard2005</td>
<td>+</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Robillard2008</td>
<td>+</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Saul2007</td>
<td>+</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Trifu2008</td>
<td>+</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Trifu2009</td>
<td>+</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### 7.5 Final Conclusions and Future Work

<table>
<thead>
<tr>
<th>Reference</th>
<th>+</th>
</tr>
</thead>
<tbody>
<tr>
<td>Petrenko2008</td>
<td></td>
</tr>
<tr>
<td>Wilson2010</td>
<td>+</td>
</tr>
<tr>
<td>Marcus2004</td>
<td></td>
</tr>
<tr>
<td>Poshvanyk2007</td>
<td>+</td>
</tr>
<tr>
<td>Gay2009</td>
<td>+</td>
</tr>
<tr>
<td>Grant2008</td>
<td></td>
</tr>
<tr>
<td>Shepherd2006</td>
<td></td>
</tr>
<tr>
<td>Hill2009</td>
<td></td>
</tr>
<tr>
<td>Abebe2010</td>
<td>+</td>
</tr>
<tr>
<td>Robillard2003</td>
<td></td>
</tr>
<tr>
<td>Eisenbarth2001</td>
<td>+</td>
</tr>
<tr>
<td>Eisenbarth2003</td>
<td>+</td>
</tr>
<tr>
<td>Koschke2005</td>
<td>+</td>
</tr>
<tr>
<td>Antoniol2005</td>
<td>+</td>
</tr>
<tr>
<td>Antoniol2006</td>
<td>+</td>
</tr>
<tr>
<td>Rohatgi2007</td>
<td>+</td>
</tr>
<tr>
<td>Rohatgi2008</td>
<td>+</td>
</tr>
<tr>
<td>Rohatgi2009</td>
<td>+</td>
</tr>
<tr>
<td>Walkinshaw2007</td>
<td>+</td>
</tr>
<tr>
<td>Poshvanyk2006</td>
<td>+</td>
</tr>
<tr>
<td>Poshvanyk2007</td>
<td>+</td>
</tr>
<tr>
<td>Liu2007</td>
<td></td>
</tr>
<tr>
<td>Asadi2010</td>
<td>+</td>
</tr>
<tr>
<td>Revele2010</td>
<td>+</td>
</tr>
<tr>
<td>Hayashi2010</td>
<td></td>
</tr>
<tr>
<td>Zhao2004</td>
<td>+</td>
</tr>
<tr>
<td>Zhao2006</td>
<td></td>
</tr>
<tr>
<td>Hill2007</td>
<td>+</td>
</tr>
<tr>
<td>Shao2009</td>
<td></td>
</tr>
<tr>
<td>Ratiu2006</td>
<td></td>
</tr>
<tr>
<td>Ratiu2007</td>
<td></td>
</tr>
<tr>
<td>Shepherd2007</td>
<td>+</td>
</tr>
<tr>
<td>Hayashi2010</td>
<td></td>
</tr>
<tr>
<td>Eaddy2008</td>
<td>+</td>
</tr>
<tr>
<td>Chen2001</td>
<td></td>
</tr>
<tr>
<td>Cubranic2003</td>
<td>+</td>
</tr>
<tr>
<td>Cubranic2004</td>
<td>+</td>
</tr>
<tr>
<td>Cubranic2005</td>
<td>+</td>
</tr>
</tbody>
</table>

176
<table>
<thead>
<tr>
<th>Paper</th>
<th>Symbol 1</th>
<th>Symbol 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cleary2007</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Cleary2009</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Wursch2010</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Ratanotayanon</td>
<td>+</td>
<td>+</td>
</tr>
</tbody>
</table>

Updated list of papers from 2011 to 2015

<table>
<thead>
<tr>
<th>Paper</th>
<th>Symbol 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ziftci2012</td>
<td>+</td>
</tr>
<tr>
<td>Yousefi2011</td>
<td>+</td>
</tr>
<tr>
<td>Heydarnoori2012</td>
<td>+</td>
</tr>
<tr>
<td>Kimmig2011</td>
<td>+</td>
</tr>
<tr>
<td>Hill2011</td>
<td>+</td>
</tr>
<tr>
<td>Hill2014</td>
<td>+</td>
</tr>
<tr>
<td>Poshyvanyk2012</td>
<td>+</td>
</tr>
<tr>
<td>Xue2012</td>
<td>+</td>
</tr>
<tr>
<td>Abebe2013</td>
<td>+</td>
</tr>
<tr>
<td>Panichella2013</td>
<td>+</td>
</tr>
<tr>
<td>Rahman2015</td>
<td>+</td>
</tr>
<tr>
<td>Kazato2013</td>
<td>+</td>
</tr>
<tr>
<td>Maras2013</td>
<td>+</td>
</tr>
<tr>
<td>Medini2014</td>
<td>+</td>
</tr>
<tr>
<td>Dit2013</td>
<td>+</td>
</tr>
<tr>
<td>Scanniello2011</td>
<td>+</td>
</tr>
<tr>
<td>Peng2011</td>
<td>+</td>
</tr>
<tr>
<td>Petrenko2013</td>
<td>+</td>
</tr>
<tr>
<td>Alhindawi2013</td>
<td>+</td>
</tr>
<tr>
<td>Dilshener2012</td>
<td>+</td>
</tr>
<tr>
<td>Nie2012</td>
<td>+</td>
</tr>
<tr>
<td>Bassett2013</td>
<td>+</td>
</tr>
<tr>
<td>Kastner2014</td>
<td>+</td>
</tr>
<tr>
<td>Scaniello2015</td>
<td>+</td>
</tr>
<tr>
<td>Wursch2013</td>
<td>+</td>
</tr>
<tr>
<td>Zamani2014</td>
<td>+</td>
</tr>
<tr>
<td>Kevic2014</td>
<td>+</td>
</tr>
</tbody>
</table>
Appendix B: Source Code
Example of Using FUSIX API

IndexAndQueryExample.java

```java
package f u s i x . c l i ;
import java . n i o . f i l e . Paths ;
import java . u t i l . List ;
import java . u t i l . Set ;
import java . u t i l . concurrent . ExecutionException ;
import java . u t i l . concurrent . Executors ;
import java . u t i l . concurrent . Future ;
import co . f u s i x . component . Component ;
import co . f u s i x . corpus . Configurations ;
import co . f u s i x . corpus . Corpus ;
import co . f u s i x . corpus . Granularity ;
import co . f u s i x . versioncontrol . Recentness ;
public class IndexAndQueryExample {
    private static final int NUM_THREADS = Runtime.getRuntime().availableProcessors();
    private static final String TEST_QUERY = "your_query_here";

    public static void main(String[] args) {
        ExecutorService executor = Executors.newFixedThreadPool(NUM_THREADS);
        Corpus<List<String>> corpus = Configurations.builder()
            .srcDir(Paths.get("your_source_directory_here"))
            .indexDir(Paths.get("your_index_directory_here"))
            .granularity(Granularity.METHOD)
            .recentness(Recentness.RECENT)
            .filtered()
            .source(Source.VCS)
            .build();
        /*
         * create corpus configuration
        */
        Future<List<String>> indexTask = executor.submit(corpus.create());
        /*
         * submit the corpus for indexing given this particular configuration
        */
        if (indexTask.isDone()) {
            try {
```
List<String> details = indexTask.get();
// print indexing details
details.forEach(System.out::println);
} catch (InterruptedException | ExecutionException e) {
    e.printStackTrace();
}

/*
 * submit query
 */
Future<Set<Component>> queryTask = executor.submit(corpus.search(TEST_QUERY));

Set<Component> components;
try {
    components = queryTask.get();
    // print out search results
    components.forEach(System.out::println);
} catch (InterruptedException | ExecutionException e) {
    e.printStackTrace();
}

executor.shutdown();
Appendix C: Screen-shot of Demo Application
1) Search query

Type in your query and search for source code components

Query: implementation of context factory

Project: Rhino

Granularity: Method

Data source: Change-sets

Recentness of change-sets: Most Recent

Filtering of change-sets: All

Search

2) Configuration

3) Results

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>src/org/mozilla/javascript/ContextFactory.java::ContextFactory_void_exit()</td>
</tr>
<tr>
<td>2</td>
<td>src/org/mozilla/javascript/Context.java::Context_Object_call(ContextAction)</td>
</tr>
<tr>
<td>3</td>
<td>src/org/mozilla/javascript/Context.java::Context_void_exit()</td>
</tr>
<tr>
<td>5</td>
<td>src/org/mozilla/javascript/ContextFactory.java::ContextFactory_Context_enter()</td>
</tr>
</tbody>
</table>
References


REFERENCES


REFERENCES


REFERENCES


Lientz, B. P. and Swanson, E. B. (1980). Software maintenance management: a study of the maintenance of computer applications software in 487 data processing organizations. 4 25


Miller, G. A. (1956). The magical number seven, plus or minus two: some limits on our capacity for processing information. *Psychological review*, 63(2):81. 5
REFERENCES


REFERENCES


REFERENCES


