MINING SOFTWARE REPOSITORIES TO SUPPORT SOFTWARE EVOLUTION

A dissertation submitted
to Kent State University in partial
fulfillment of the requirements for the
degree of Doctor of Philosophy

by
Huzefa Kagdi

August 15, 2008
Dissertation written by

Huzefa Kagdi

M.S., Kent State University, USA, 2003

B.E., Birla Vishwakarma Mahavidyalaya, India 1998

Approved by

Dr. Jonathan I. Maletic, Chair, Doctoral Dissertation Committee

Dr. Yuri Breitbart, Members, Doctoral Dissertation Committee

Dr. Gregory M. Shreve

Dr. Ruoming Jin

Dr. Michael L. Collard

Accepted by

Dr. Robert A. Walker, Chair, Department of Computer Science

Dr. Timothy Moerland, Dean, College of Arts and Sciences
TABLE OF CONTENTS

LIST OF FIGURES ........................................................................................................ IX

LIST OF TABLES ........................................................................................................ XI

ACKNOWLEDGEMENTS ............................................................................................... XIV

CHAPTER 1 INTRODUCTION ......................................................................................... 1

1.1 The Philosophical Take on Software Changes ....................................................... 1

1.2 Conducted Research .............................................................................................. 3

1.3 Contributions ........................................................................................................ 5

1.4 Organization .......................................................................................................... 6

CHAPTER 2 A PRELUDE TO MINING SOFTWARE REPOSITORIES ..................... 7

2.1 Scope ..................................................................................................................... 8

2.2 Historical Perspective to MSR Research .............................................................. 9

2.3 Previous Classifications in MSR .......................................................................... 10

2.4 Organization ......................................................................................................... 12

2.5 Dimensions of the Survey ................................................................................... 12

   2.5.1 Information Sources ..................................................................................... 14

   2.5.2 Purpose ......................................................................................................... 17

   2.5.3 Methodology ................................................................................................. 18

   2.5.4 Evaluation .................................................................................................... 20

2.6 A Taxonomy of MSR Approaches ....................................................................... 22

   2.6.1 A Layered Taxonomy .................................................................................. 22

   2.6.2 Validating the Taxonomy .......................................................................... 24
2.7 MSR – A Comprehensive Survey .................................................................26
2.7.1 Metadata Analysis ..................................................................................27
2.7.2 Static Source Code Analysis .................................................................44
2.7.3 Communication Via Source Code Comments ........................................51
2.7.4 Source Code Differencing and Analysis .............................................51
2.7.5 Software Metrics ..................................................................................59
2.7.6 System Complexity .............................................................................64
2.7.7 Visualization .........................................................................................66
2.7.8 Clone-Detection Methods .................................................................73
2.7.9 Frequent Pattern Mining .................................................................77
2.7.10 Information Retrieval (IR) Methods ...............................................82
2.7.11 Classification with Supervised Learning .......................................90
2.7.12 Social Network Analysis ...............................................................92
2.8 Discussion & Open Issues in MSR Research ........................................95
2.8.1 MSR on Fine Grained Entities .......................................................96
2.8.2 Historical Context – How many Versions? ..................................98
2.8.3 Threats to Validity in MSR .............................................................99
2.9 Summary ...............................................................................................100

CHAPTER 3 UNCOVERING TRACEABILITY LINKS FROM COMMITS IN
SOFTWARE REPOSITORIES ........................................................................106

3.1 Open Source and Traceability ...............................................................108
3.2 Uncovering Traceability Links ...............................................................110
CHAPTER 5 COMBINING SINGLE-VERSION AND EVOLUTIONARY DEPENDENCIES FOR CHANGE PREDICTION

5.1 IA and MSR ................................................................. 149
5.2 Supporting a Combined Approach ........................................ 152
5.3 Evaluation Framework ...................................................... 153
5.4 Summary ......................................................................... 157

CHAPTER 6 MINING EVOLUTIONARY DEPENDENCIES FROM WEB-LOCALIZATION REPOSITORIES

6.1 Document Localization: the GNU Model .................................. 162
6.2 Evolution of Translated Documents ......................................... 165
6.3 Automatically Uncovering Evolutionary Couplings .................... 172
6.4 Document-Change History ................................................... 172
6.5 Uncovering Evolutionary Dependencies ................................... 173
6.6 Example ........................................................................... 178
6.7 Evaluation ........................................................................ 180
6.8 Document Repository and Tool Set ........................................ 181
6.9 Mined Patterns and Their Characteristics ................................ 183
6.10 Validating Support for Evolution .......................................... 188
6.11 Threats to Validity ............................................................ 193
6.12 Related Work .................................................................. 196
6.13 Summary ........................................................................ 198
8.3.3 Using History to Examine Variants .................................................243

8.4 Threats to Validity ..................................................................................247

8.5 Related Work ............................................................................................248

8.6 Summary ...................................................................................................250

CHAPTER 9 CONCLUSIONS AND FUTURE WORK .....................................252

REFERENCES .................................................................................................255
LIST OF FIGURES

Figure 1. Four-Layer Taxonomy of MSR Approaches. .................................................. 105

Figure 2. Histogram of the number of different file types with regards to the number of change-sets (i.e., commits) in the six open source systems. It shows two or more file types substantially co-occur in the change-sets in all these systems. ......................... 113

Figure 3. A Snippet of kdelibs Subversion Log. .......................................................... 116

Figure 4. Online manuals for the KDE application KSpell in languages a.) US English and b.) German. .............................................................................................................. 162

Figure 5. Excerpts of a.) the base-document man-kdeoptions.7.docbook from kdelibs in US English and d) its corresponding translated-document in German. Excerpts of of b.) internationalized-document for translation to other languages (POT file) and c.) localized-document in German (PO) used in the translation process. ................. 166

Figure 6. a.) An example of four revisions available directly from a web-localization repository and b) the change-set-oriented history converted to a document-history. ................................................................................................. 171

Figure 7. Transactions in sqminer’s input format. Each event is specified on a separate line and consists of a document name along with its path, version-number, and the set of documents co-changed. .................................................................................. 177

Figure 8. The structure of the KDE localization repository. ............................................ 180

Figure 9. a.) Distribution of patterns with regards to the number of versions taken to complete changes in them and b.) frequency distribution of pattern sizes.............. 183
Figure 10. The number of common patterns that are found in the number of documents-histories.

Figure 11. a.) The number of common patterns found in the histories of different languages, b.) total number of supported patterns by the translation history of a specific language.

Figure 12. A screen shot of the KDE web page showing the status of translated help documents in the trunk of the l10n repository ordered by language on February 3, 2007.

Figure 13. An example of four function definitions with calls to functions $a$, $b$, and $c$ for demonstrating patterns, variants, and violations.

Figure 14. Variants per pattern showing that sequential-pattern mining outperforms itemset mining as the ratio of the number of variants per pattern is always low for ordered patterns.

Figure 15. Violations per function with variants showing that sequential-pattern mining outperforms itemset mining for lower support values and performs equally for higher support values.
LIST OF TABLES

Table 1. The subset of surveyed approaches that study changes to artifacts .............. 101
Table 2. The subset of surveyed approaches studying changes to properties .............. 102
Table 3. The subset of surveyed approaches studying changes to both artifacts and properties. ................................................................. 103
Table 4. The approaches surveyed and organized by the MSR tasks they address. The tasks are categorized into related groups .......................................................... 104
Table 5. Statistics of the change-sets analyzed for different types of files in six open source systems .................................................................................. 114
Table 6. Groups formed from the change-sets extracted from the KDE repository by the different heuristics ................................................................. 127
Table 7. Evolutionary couplings (EC) and traceability couplings (TC) uncovered from the KDE repository. All patterns have a minimum support of five. .............. 127
Table 8. The minimum, maximum, and average number of files in the traceability couplings in the training-set ........................................................................ 128
Table 9. Traceability couplings (TC) from the KDE repository that are used as evaluation data-set for the traceability couplings (TC) in Table 7. All patterns have a minimum support of five. ...................................................... 131
Table 10. Coverage and Recall for the evaluation-set (Table 7) compared against the training-set (Table 9) .................................................................................. 131
Table 11.  Precision for the evaluation-set (Table 7) compared against the training-set (Table 9)........................................................................................................................................................................133
Table 12.  Patterns produced by sqminer from the transactions of Figure 4. ...............179
Table 13.  Accuracy of patterns with different history period and mining parameter combinations........................................................................................................................................................................195
Table 14.  Linux call sequence statistics. ..............................................................................................................................212
Table 15. A comparison of sequential-pattern mining (ordered) and itemset-pattern mining (unordered) approaches for Linux kernel v2.6.14. Violations are the total number of (variant, function) pairs. ..........................................................................................................................214
Table 16.  Apache httpd Call Statistics............................................................................................................................236
Table 17.  The number of candidate ordered patterns, rules, variants, and violations in Apache httpd (v2.0.55). Candidates are categorized based on the inclusion of syntactic context........................................................................................................................................................................237
Table 18.  The number of candidate functions that follow rules and contain variants. These functions are grouped based on the categorization of the rules. An interesting observation is that there is a number of functions containing both categories of rules (and also variants) ........................................................................................................................238
Table 19.  Comparison of call-usage rules between v2.0.55 and v2.2.0. The rules are accurately mined as they are retained between versions. ........................................241
Table 20. Comparison of call-usage rules between v2.0.55 and v2.2.0. The functions are accurately identified as they obey rules in both versions. ..................................................242
Table 21. Comparison of variants of call-usage rules between v2.0.55 and v2.2.0. The results show that our approach correctly identified a number of them. .......................... 244

Table 22. The functions with variants in v2.0.55 that are fixed in v2.2.0 due to a variant becoming a rule................................................................. 247
ACKNOWLEDGMENTS

I express my deepest gratitude to my beloved wife, Shehnaaz, and my wonderful parents, Zarina and Hatim, for their enormous support, infinite patience, and unwavering belief towards me, as always. My loving brother and sister, Husein and Tasneem, have always been on my side. There is a litany of family members and friends who are not individually mentioned here, but they certainly made a difference.

I was fortunate to have come across many great individuals during my journey of graduate studies. My research advisor, Dr. Jonathan I. Maletic, was always there in times of great needs and deeds. I have collected many cherished moments and unique experiences from colleagues at Software Development Laboratory (<SDML>) and Kent State University. I also greatly appreciated the service of my dissertation committee.

Thank you one and all.

Huzefa Kagdi

June 30, 2008, Kent, Ohio
CHAPTER 1

Introduction

Software artifacts such as source code and design documents are produced in an inherently incremental manner via continuous change. They undergo changes due to factors such as defect corrections, feature additions, and design improvements. This general phenomenon is described as software evolution [Bennett, Rajlich, 73-87 2000; Lehman, Ramil 2001c; Lehman, Belady 1985a; Rajlich et al. 2001]. Software evolution is a particularly complex phenomenon in case of long-lived, large-scale systems. It is not uncommon for a long-lived, large software system to progress through years of development history, a number of developers, and millions of lines of code. Therefore, realizing even a tad bit of change in such a large-scale software system may not be always straightforward.

1.1 The Philosophical Take on Software Changes

Clearly, changes are the central force driving software evolution. Therefore, it is not surprising that a paramount effort has been (and should be) devoted in the software engineering community to systematically understand, estimate, and manage changes to software artifacts. This includes forward engineering (the initial piece of software/code), program comprehension (help in understanding existing code, which becomes increasingly difficult with the increasing size of code base), concept location (where a particular functionality is implemented in a code – a starting point of a change), Impact
analysis (identify change-prone software entities), reverse engineering (produced a high level abstraction to give a different view with the goal of aiding program comprehension), and requirements engineering (including verification and validation.) Understanding and managing software changes are, in many ways, about identifying, expressing, understanding, and managing dependencies or couplings among software entities at both intra and inter artifact types (e.g., call, data, and control dependencies in source code, and source code and user documentation.) Traditional software engineering paradigm considers couplings in a unilateral view, typically the latest version, of the software artifacts or system. The basic premise is to use software dependencies between software entities, in a current version, as an estimation of their future change proneness. The facts and knowledge related to the actual changes, occurred during the evolution, are largely ignored in the notion of software couplings as it pertains to software evolution and changes. The philosophical nature of this traditional paradigm seems to align with the deductive method – logic, alike mathematical axioms and theorems, is the driving authority. In that, the logic statement is that the presence of software dependencies among software entities implies also their change dependencies. For example, given a call dependency between two functions, a change in the called function would also imply a change in the caller function. Therefore, understanding, estimating, and managing proposed or future changes reduce to a logical derivation for all (and only) the software entities directly or indirectly related via dependencies (e.g., of static and/or dynamic type.)
A new paradigm, rooted in the area of *Mining Software Repositories* (MSR), is to analyze multiple versions, i.e., actual past changes in software repositories such as *Subversion* and CVS, to identify for dependencies between software entities that are found to co-change. Such dependencies that are based on the actual evolution history of a software system are termed as *evolutionary dependencies or couplings* [Gall, Hajek, Jazayeri 1998; Kagdi, Collard, Maletic 2007a; Zimmermann et al. 2005]. Evolutionary couplings between software entities are considered as an indicator of their future change proneness. This brings forward a new dimension of historical context in the development of software engineering and evolution tools that was missing previously. The philosophical nature of this new paradigm seems to align with the *inductive or scientific method* – observations from natural, real phenomenon is the driving authority. In that, the changes observed in the past evolution of a specific system is used as a basis to speculate the change dependencies between any given software entities. For example, if two files or functions are observed to co-change in a number of previously submitted patches or change-sets, an inherent change dependency between them is surmised. Therefore, understanding, estimating, and managing proposed or future changes become a discovery process for uncovering patterns, trends, and relationships from past changes.

### 1.2 Conducted Research

The thesis develops an approach that mines evolutionary couplings from the version histories of a software system. A combination of lightweight source-code analysis/differencing, development heuristics, and sequential-pattern mining technique are used in the mining process of evolutionary couplings. These evolutionary couplings
are identified between different artifact types (e.g., source code and documentation, and end-user documentation in different natural languages) [Kagdi, Maletic 2007b; Kagdi, Maletic, Sharif 2007], as well as at various fine-grained representations (e.g., methods, control statements, preprocessor, and comments) of source code [Kagdi 2007], from the commits stored in software repositories. The applications of the mined evolutionary couplings are shown on a number of software evolution tasks such as source code change and impact analysis, uncovering software traceability links, and software document localization process. Furthermore, a hybrid software-change prediction model [Kagdi, Maletic 2007a] is developed that combines the change-sets estimated from software dependency analysis (via program analysis techniques of a single version) and the evolutionary couplings uncovered from software version histories (via multiple-version analysis.) The end goal of these research efforts is to build a holistic software-change recommendation approach that includes the historical component of software evolution to support the impact analysis task.

Finally, the developed approach is used to automatically mine latent programming rules from a body of large source code has been developed [Kagdi, Collard, Maletic 2007b; c]. These programming rules consist of function-call usages and their surrounding syntactic constructs (e.g., if-statements.) Additionally, non-standard or error-prone variant usages, i.e., violations, of these rules, such as those with missing or out of order calls, are automatically identified along with their specific contextual location.
Large-scale software systems with a diverse range of application/solution domains such as KDE (~500 KLOC), Linux kernel (~600 KLOC), and Apache web server are used in the evaluation process. Our general evaluation methodology with regard to uncovering traceability links and document localization is to first mine a portion of the version history for evolutionary couplings. We call this the training-set. Next we mine a later part of the version history called the evaluation-set and see if the results generated from the training-set can accurately predict changes that occur in the evaluation-set. Similar approach along with manual examination is used in the evaluation of call-usage patterns and their violations. The results on a number of versions of KDE show that the evolutionary couplings uncovered by our approach are quite precise in predicting future changes or couplings in the context of traceability links and localized documents. Additionally, the results of the evaluation on Linux kernel and Apache httpd show that our approach is able to uncover latent programming rules and their violations that a previous approach cannot (and did not.)

1.3 Contributions

To the best of our knowledge, this is the first work in MSR that uncovers ordered evolutionary couplings including traceability patterns, provides fine-grained source code granularity up to statement and comment levels, and change predictions over multiple future versions and not just the immediate next version. There is no other work in the literature to uncover evolutionary couplings in internationalized and localized software documents from the document-localization repositories. Furthermore, our work is the first one to uncover call-usage patterns with both call order and syntactic context from
source code in the context of latent programming rules. Finally, an in-depth survey and the very first taxonomy of MSR approaches have resulted as a byproduct.

1.4 Organization

The rest of the document is organized as follows. A comprehensive survey and taxonomy of MSR is provided as a background material in CHAPTER 2. Mining evolutionary couplings in the context of traceability-link recovery is presented in CHAPTER 3. Mining of evolutionary couplings at fine-grain granularities from source code is discussed in CHAPTER 4. CHAPTER 5 presents a hybrid model for software-change prediction. CHAPTER 6 shows the application of evolutionary couplings in the context of document internationalization and localization. Two approaches to mining call-usage patterns and their violations are compared in CHAPTER 7, and an extension of our approach with syntactic context is presented in CHAPTER 8. Finally, we conclude in CHAPTER 9 along with the discussion on open issues and future direction.
CHAPTER 2
A Prelude to Mining Software Repositories

The term Mining Software Repositories (MSR) has been coined to describe a broad class of investigations into the examination of software repositories. Here software repositories refer to artifacts that are produced and archived during software evolution. They include sources such as the information stored in source code version-control systems (CVS), requirements/bug-tracking systems (Bugzilla), and communication archives (e.g., email.) These repositories hold a wealth of information and provide a unique view of the actual evolutionary path taken to realize a software system. Often this data exists for the entire duration of a project and can represent thousands of versions with years of details about the development. This data includes such things as individual versions of the system, the changes, and metadata about the changes (e.g., who made the change, why the change was made, when the change was done, etc.)

Software engineering researchers have devised and experimented with a wide spectrum of approaches to extract pertinent information and uncover relationships and trends from repositories in the context of software evolution. This activity is analogous (but not limited) to the field of data mining and knowledge discovery, hence the term MSR. The premise of MSR is that empirical and systematic investigations of repositories will shed new light on the process of software evolution and the changes that occur over
time by uncovering pertinent information, relationships, or trends about a particular evolutionary characteristic of the system.

A comprehensive literature survey on approaches for Mining Software Repositories (MSR), in the context of software evolution, was conducted during the prologue of this work. In particular, the survey deals with those investigations that examine multiple versions of software artifacts or other temporal information. A taxonomy is derived from the analysis of this literature and presents the work via four dimensions: the type of software repositories mined (what), the purpose (why), the adopted/invented methodology used (how), and the evaluation method (quality.) Next, the survey and taxonomy are detailed.

2.1 Scope

There is a wide range of research investigations that apply mining techniques to software artifacts. Some examine just a single version of an artifact while others examine the entire version history of a software system. Here we limit the scope of our survey to only those research investigations that examine multiple snapshots of software artifacts (e.g., source code version from CVS, system release, etc.) and/or other temporal information (e.g., effect on size and structure of a system, bug reports, etc.) Our goal is to survey the literature that specifically investigates evolutionary changes of software artifacts. Additionally, our survey only covers works published before August 2006.

Research and approaches that primarily examine a single version or release of a software system are excluded from this survey, as they typically do not directly address the issues of software evolution and change. For example, we felt work that focused on
analyzing a single version, and just happened to use a data-mining technique to do
analysis, is not within the scope of this survey. This type of investigation is research on
analysis methods to support testing (or some other software engineering task.) In other
words, this is not a survey of investigations applying data-mining techniques to software
engineering problems but rather a survey of investigations that examine the changes and
evolution of software and use data mining and other similar techniques. In a very few
cases, we have included work that presented techniques that could readily be applied to
multiple versions but was only applied to a single version. These are included for
completeness and typically represent important contributions to the study of software
repositories.

2.2 Historical Perspective to MSR Research

Historically, there have been a number of efforts to examine long-term software-
project data to better understand software evolution. Lehman et al. [Lehman 1980;
Lehman, Perry, Ramil 1998; Lehman, Ramil 2001a; b; Lehman, Belady 1985b; Ramil,
Lehman 2000] reported various results on the software change and nature of software
evolution between 1969 and 2001 based on long-term studies of several IBM products.
The most notable results of these types of studies are the laws of software evolution
[Lehman 1980; Lehman, Perry, Ramil 1998; Lehman, Belady 1985b], metrics of software
evolution [Ramil, Lehman 2000], classification of programs [Lehman, Ramil 2001b], and
a theory of software evolution [Lehman, Ramil 2001a]. Weiss and Basili [Weiss, Basili
1985] collected and analyzed software data (including changes) from multiple software
systems as they were developed. Eick et al. [Eick et al. 2001] observe the phenomenon
of code decay (i.e., changes to a system become difficult in terms of cost, time, and quality over its lifetime) by leveraging software repositories.

In the past, MSR investigations were almost always subjected on industrial systems. Consequently, research efforts were limited to a select few software systems (and application domains) or hampered by the lack of historical software data that was publicly available. Recently, there has been a rapid (and important) paradigm shift with regards to the above situation, mostly attributed to the establishment and wide prevalence of open-source software development. Arguably, the open-source paradigm has been successful in producing numerous high-quality projects that continue to live and evolve.

Given the recent large influx of MSR investigations, it has now become imperative to show the similarities and variations among these approaches in the context of software evolution. This is important in order to appreciate the contributions of these MSR investigations with respect to which purposes or aspects of software evolution they support. However, there is no common nomenclature in terms of reference model, classification, and/or process model to form a basis for describing the overall MSR investigation in the context of software evolution.

2.3 Previous Classifications in MSR

Little effort has been made in the direction of comparing and contrasting MSR approaches. Besides our own initial survey [Kagdi, Collard, Maletic 2005], which examined six approaches, only two other brief surveys have been presented. German et al. [German, Cubranic, Storey 2005] describe a framework that classifies three MSR tools with regards to support for different types of user roles (e.g., maintainer,
researcher), information sources accessed and utilized, and infrastructure needed for integration, organization, and analysis of the collected data/information. As such their work focuses on comparing the usability and the underlying infrastructure of various tools supporting MSR. In contrast, the survey presented here is targeted at describing MSR approaches and the different reasons for mining. In the other work, Kim and Notkin [Kim, Notkin 2006] surveyed program matching (i.e., differencing) techniques with regards to the supported granularity (e.g., line and functions), program representation (source code, AST, and control-flow graphs), and underlying comparison method (e.g., name similarity.) Program matching is described as comparing the elements between two versions (e.g., added, deleted, or renamed lines.) They evaluate the surveyed techniques with two synthetic change scenarios (combinations of add, move, split, and rename.) The goal was to provide assistance to researchers in choosing the appropriate differencing technique. Our goal is much broader than comparing alternative techniques (tools) for a problem (i.e., program differencing or change management.) Additionally, we are interested in the types of artifacts (not just source code/programs), types of questions (for various purposes), and methodologies researchers have investigated in MSR.

In other, less related work, Buckley et al. [Buckley et al. 2005] presents a taxonomy of software changes from a perspective of software-evolution tools. Their taxonomy consists of four dimensions: temporal properties (e.g., compile-time), object of change (e.g., file and executable code), system properties (e.g., system needs to be up), and degree of automation (e.g., partial and manual.) Their taxonomic descriptions are
applied to the tools Refactoring Browser, CVS, and eLiza. Our interest is not just limited to what software-change support is directly available from software-change management tools, but how researchers are utilizing and extending this information for answering MSR questions.

2.4 Organization

The work presented here has two main contributions. The first is a comprehensive survey of MSR approaches, in the context of software evolution, and the second is a derived taxonomy of those approaches. In the next section (2.5) we present the dimensions of the survey, that is, the main characteristics of the literature we survey. We consider approximately 80 approaches from papers presented in the literature that meet this criterion. This describes how the survey is organized and sets the stage for the resulting taxonomy. Following that in Section 2.6 is our layered taxonomy and its validation (Section 2.6.2.) Section 2.7 is an in-depth discussion of the published literature in MSR. All of these surveyed approaches are presented with respect to our taxonomy. Section 2.8 provides a discussion on identified open issues in the MSR research followed by our conclusions.

2.5 Dimensions of the Survey

We conducted an initial literature survey [Kagdi, Collard, Maletic 2005] with the goal of defining a taxonomy of MSR approaches. Here, we greatly extend that work and present a broader survey and analysis of a large number of MSR investigations. Our search space of literature includes works from MSR-specific venues including the
ACM/IEEE Workshop on Mining Software Repositories (MSR) that took place in 2004, 2005 and 2006 along with a special issue of *IEEE Transactions on Software Engineering (TSE)* on Mining Software Repositories that appeared in July 2005. However, MSR research has much older roots and broader interests than these recent specific venues. A number of established venues in the software engineering and evolution community including ACM/IEEE International Conferences on Automated Software Engineering (ASE), Software Engineering (ICSE), and Software Maintenance (ICSM) regularly publish MSR types of investigations. As such our list provides a wide spectrum of MSR research.

From a thorough examination of the literature surveyed we identified four dimensions in order to objectively describe and compare the different approaches. These dimensions are used as sampling criteria in the collection and analyses of the literature. The dimensions are:

- The software repositories utilized – What information sources are used?
- The purpose of MSR – Why mine or what to mine for?
- The methodology – How to achieve the purpose of mining from the selected software repositories?
- The evaluation of the undertaken approach – How to assess quality?

At this point we do not imply any specific explicit order, priority, or role to the four dimensions. This can be attributed to the lack of a defined process model for MSR. Here, the order in which these dimensions are presented may not depict the order of a typical MSR process. Let us now discuss these dimensions in more detail.
2.5.1 Information Sources

A fundamental question is what types of sources can be considered as software repositories? Recent literature highlights source-control systems, defect-tracking systems, and archived communications as the main data sources for MSR investigations. Source-control systems are primarily used for storing and managing changes to source code artifacts, typically files, under evolution. Defect-tracking systems are used to manage the reporting and resolution of defects/bugs/faults and/or feature enhancements. Archived communications such as email store discussions between project participants, making them sources for information including change rationales.

Clearly, these types of software repositories vary in their usage, information content, and storage format. Furthermore, these repositories are managed and operated (for the most part) in isolation and have no explicit direct relationship with each other. For example, no explicit information is typically maintained between a particular “bug” in the defect-tracking system and the corresponding source code changes in the source-control repositories. A number of approaches have been proposed to integrate the various software repositories into a common information source, typically as a relational database [Alonso, Devanbu, Gertz 2004; Conklin, Howison, Crowston 2005; Gasser, Ripoche, Sandusky 2004; German 2004b; Robles, González-Barahona, Ghosh 2004; Zimmermann et al. 2004], and to access data from software repositories that are not directly available (e.g., webpage scraping [Howison, Crowston 2004] of a defect-tracking system maintained by Bugzilla.) These approaches are not included in the survey since they only change how data is obtained, not the purpose, method, or evaluation of the MSR
approach. They may make the task of completing an MSR investigation easier, but they do not change what investigations can be performed.

Nonetheless, these repositories have a common goal of supporting software evolution by managing the lifecycle of a software change. We define a *software change* as an addition, deletion, or modification of any software artifact (e.g., requirement specification, design documents, and test cases) such that it alters, or requires amendment of, the original assumptions of the subject system. The typical realization of a software change is a modification to the source code. We typically consider a new version to be created when a source code change occurs. Therefore, the fundamental unit of software evolution is the source code change. All other information is maintained to help understand, rationalize, and manage source code changes.

In light of the primacy of source code change, we see three basic categories of information in a software repository that can be mined:

- The software artifacts/versions
- The differences between the artifacts/versions
- The metadata about the software change

Our survey will show that most of the source code repositories being examined are managed by *Concurrent Versions System - CVS* (www.cvshome.org.) In addition to storing differences across document versions, *CVS* augments this with metadata such as commit comments, user-ids, timestamps, and other similar information. This metadata describes respectively the *why*, *who*, and *when* context of a source code change. *CVS* is completely ignorant of the underlying syntax and semantics of the source code. The
differences between source code documents are stored as physical entities (file and line numbers) and not in terms of the entities inside a file, e.g., function or statement, that are more familiar to a developer or MSR. Moreover, CVS suffers from other management limitations and irregularities. It does not maintain the grouping of several changes in multiple files (deltas) as a single logical change (transaction.) Therefore, the original commit operations performed by developers are lost. Also, CVS does not maintain explicit branch and merge points. To deal with these problems, a number of variants of sliding and fixed window methods have been proposed to approximate CVS commits and transactions from the deltas [Gall, Hajek, Jazayeri 1998; German 2004a; Zimmermann et al. 2004]. Also, various branch and merge point detection algorithms have been described [Chen et al. 2001; Fischer, Pinzger, Gall 2003; Zimmermann et al. 2004; Zou, Godfrey 2003]. These issues are important and may have potential impact on a MSR investigation. However, the papers addressing these issues are excluded from the survey as they are more suited for a discussion on software configuration research issues; rather than MSR research. More modern version-control systems such as Subversion (subversion.tigris.org) offer a step forward on the above issues eliminating a number of major roadblocks in MSR research (and version-control usage.) For example, Subversion preserves the atomicity of commits, i.e., all files committed together as a single change-set, and thus eliminates the change-set recovery effort (which is typically done for a CVS repository in MSR.)

Additional metadata regarding a source code change is available from other types of software repositories. Bugzilla (www.bugzilla.org) is a defect/bug tracking system
that maintains the history of the entire lifecycle of a bug (or a feature.) Each bug is maintained in the form of a record, termed a bug report. In addition to storing the description of a bug, it includes monitoring fields such as when a bug was reported, assignment to a maintainer, priority, severity, and current state (open/closed.) Archived communications in the form of email lists capture discussions between developers over the lifetime of the project.

In summary, metadata forms a valuable source for deriving high-level semantic information in the context of software change. It can be analyzed independently of the other data, or richly combined with the source code and difference information.

2.5.2 Purpose

Researchers mine data and metadata in a software repository to extract pertinent information and/or uncover relationships or trends about a particular evolutionary characteristic. For example, one may be interested in the growth of a system, change relationship between source code entities, or reuse of components. In order to qualitatively study a particular characteristic, and define the scope and context of the mined information, the purpose is typically expressed as a set of questions. Therefore, the purpose of mining reduces to what questions can be answered by MSR. We term these MSR questions.
Broadly speaking, there are two classes of MSR questions. The first is the market-basket\(^1\) question formulated as: If \(A\) occurs than what else occurs on a regular basis? The answer is a set of rules or guidelines describing situations of trends or relationships. For example, if \(A\) occurs then \(B\) and \(C\) happen \(X\) amount of the time.

The second type of MSR question relates to prevalence. Instances include metric and boolean queries. For example, was a particular function added/deleted/modified? Or how many and which of the functions are reused? The questions asked indicate the purpose of the mining approach.

### 2.5.3 Methodology

Given a software repository and purpose, a method must be adopted or devised to answer MSR questions. A wide spectrum of approaches ranging from conventional software engineering methods to established methods from other domains have been applied to MSR investigations.

Broadly, there are two basic strategies that can be taken. Each version may be extracted, properties computed on each version separately, and then the individually computed properties compared. This strategy corresponds to the indirect (or external) measurement and analysis of software evolution. For example, metrics for software

\(^1\) The term market-basket analysis is widely used in describing data-mining problems. The famous example about the analysis of grocery store data is that “people who bought diapers oftentimes bought beer”.
complexity, defect density, or maintainability can be computed for two versions of a system taken from CVS and the quality of the evolved system assessed. In this approach the interest is in the changes of high-level or global properties of a software system under evolution. We refer to this group of MSR investigations as interested in changes to properties.

The second perspective represents investigations that study the actual mechanisms or facts that take a software system from one version to the next. Here the focus is on the specific differences between versions. These types of approaches use the difference data supplied by CVS or other tools. This strategy corresponds to the direct (or internal) measurement and analysis of software evolution. We refer to this group of MSR investigations as interested in changes to artifacts. There can be a significant level of variation with respect to the granularity and type of source code change. Types of source code entities include physical, syntactic, documentary, etc. Likewise, one can examine changes to a file, class, or function. These differences are reflected in how sophisticated the tools used in the investigation are with respect to such things as programming-language knowledge.

Researchers utilize software repositories in multiple ways. The most straightforward is to directly use the functionality of source code repositories (i.e., CVS commands) to get a particular version of the code. The individual versions and corresponding metadata can then be used to answer the MSR questions of interest using the adopted/invented methodology. Some researchers limit their study to the metadata that is directly available from the repositories. This type of metadata is analyzed to filter
the differences and source code in a semantic manner. For example, the CVS comments and the textual description of a related bug report in Bugzilla can be used to categorize the source code changes as an attribute of corrective-maintenance activity. Going a step further, the data and metadata directly available from CVS can be processed to facilitate fine-grained source code difference analysis. This allows addressing MSR questions in a source code aware manner, i.e., in terms of syntax and semantics of the programming languages.

2.5.4 Evaluation

The realm of open-source development gives us the luxury of publicly-available software repositories for many projects. SourceForge (sourceforge.net) is a well-known and widely-used site housing the software repositories of over 100,000 projects. These projects vary in size, number of contributors, application domain, and solution domain. Such a wide spectrum of repositories enables researchers to conduct empirical studies to evaluate a MSR approach. However, with this luxury comes the additional responsibility of selecting the appropriate project repository. This is important in order to validate the hypotheses, interpret results, and draw conclusions to other systems or other points in the project history in an unbiased way. In particular, the repositories of the open-source projects such as KDE, GCC, Apache, Eclipse, jEdit, and ArgoUML have been studied in multiple MSR investigations.

All MSR approaches share a common goal of utilizing the history of software projects in order to improve future evolution of the subject software system. Therefore the quality of a MSR approach with regards to improving software evolution must be
evaluated. Once again, the history in the software repositories can be used in empirical validation. A part of the history\(^2\) is normally used to develop the models and a later part of history is then used for evaluation. Two assessment metrics, *precision* (i.e., how much of found information is relevant) and *recall* (i.e., how much of all the relevant information is found) borrowed from the information-retrieval community, are widely used to evaluate MSR tools. Another approach to evaluation is to take an information-theoretic approach for evaluating probabilistic models. This has been used by Askari and Holt [Askari, Holt 2006] for predicting changes and bugs in software files. A predictive model is evaluated by comparing the distribution of predicted values (e.g., changes in files) with the true distribution (i.e., the actual observations.) The closer the predictive distribution of a model is to the true distribution, the more effective it is. Entropy measurements are typically used as a metric of comparison.

Later we will see in Section 2.7 that there is little variation in the types of software repositories and evaluation methods used in the investigations surveyed. However, there is wide variation in the methodology and the purpose of the approaches. The next section presents a taxonomy of the literature we surveyed.

\(^2\) In the rest of the discussion, we mean a portion of the history when we refer to the history of any project unless specified otherwise.
2.6 A Taxonomy of MSR Approaches

The investigations described in the surveyed papers (Section 2.7) have a number of common characteristics. They all are working on version-release histories, all work at some level of software granularity (e.g., system, subsystem, file, class, and function), and most ask very similar types of (MSR) questions. We also see that the MSR process is to extract pertinent information from repositories, analyze this information, and derive conclusions within the context of software evolution.

2.6.1 A Layered Taxonomy

In order to see the similarity and differences across the various MSR investigations, we present a layered taxonomy shown in Figure 1. This taxonomy was developed by identifying the ubiquitous traits found in all the surveyed literature. Based on the discussion and analyses in Section 2.7 with regards to the dimensions described in Section 2.5, we observed that the representation of any given MSR approach can be generalized by a four-layer taxonomic description: Software Evolution (Layer 1), Purpose (Layer 2), Representation (Layer 3), and the Information Sources (Layer 4.) We now further describe these layers.

Again, the goal of MSR is to learn more about software evolution and as discussed in Section 2.5.3 a given MSR investigation, implicitly or explicitly, is interested in the change characteristics of the high-level properties of a software system, the more detailed change in the actual artifacts, or both. Therefore, these elements are positioned in the top layer.
Researchers study the change aspects of properties and artifacts for a variety of purposes. In order to facilitate a qualitative and objective investigation, the purpose(s) is transformed into a set of Market-Basket type of MSR questions, prevalence type of MSR questions, or both. Layers 1 and 2 define the overall context in which a particular MSR investigation is conducted, or an adopted/invented MSR methodology is evaluated.

The MSR questions are answered by utilizing the three main information sources: software artifacts, their differences, and metadata about these artifacts/differences. Most repositories provide direct access to the information sources, namely the source code files, differences, and the differences metadata. However, information sources depicting high-level abstractions such as design models and architecture models are typically not directly available in the software repositories. They may need to be reverse engineered or computed to support the corresponding MSR questions. Therefore, Layer 4 represents the information sources that are readily available in the software repositories and those that need to be made available to support the MSR investigation.

Layer 3 (Representation) refers to the type (e.g., physical), granularity (e.g., system, files, classes), and expression of the artifacts and their differences. As discussed in Section 2.5.1, source code repositories are typically limited to the physical-level representation of source code (i.e., file and line numbers.) As such, the answers to MSR questions can be further extended to more fine-grain representations of artifacts and differences. Therefore, the representation of the information in the repositories can be refined based on the syntax and semantics of the underlying programming language(s.)
2.6.2 Validating the Taxonomy

We now show that our taxonomy is expressive (i.e., ability to represent a wide spectrum of MSR approaches) and effective (i.e., facilitates comparison of MSR approaches.) We represent a number of MSR approaches that will be discussed in Section 2.7 with regards to our proposed taxonomy. Tables 1-3 show the taxonomic description of the considered approaches. The tables are organized by what the approach is studying. Table 1 lists approaches studying artifacts, Table 1 lists approaches studying properties, and Table 3 lists those that study both. The first three columns represent the remaining three layers of the taxonomy and the last three columns show the main dimensions used in the survey. We also included the specific task that each approach addresses. This gives context to the general MSR question being asked. For example, the specific task of detecting evolutionary couplings between source code entities can be phrased in terms of a market basket analysis type question: What are the source code entities that are typically found together in commits? Our presentation is centered with respect to the various techniques used and/or validated for evolutionary tasks in the context of MSR investigations.

The methodologies are ordered with conventional software engineering methods appearing first, followed by those successfully adopted from other disciplines, and lastly novel experimentation. Researchers have utilized each of these methodologies to perform a variety of tasks. This is clearly evident from examining the first and last three columns of Tables 1-3. Furthermore, on examining the values of the considered cases, it is evident that MSR investigations do not enforce any rigid hierarchy or constraint on the
underlying approaches but are flexible. For example, it is not the case that the approaches studying changes of properties, only ask a particular type of MSR questions, utilize only a fixed (sub)set of artifacts/repositories (e.g., Metadata-CVS), and only consider a certain level of granularity (e.g., files.) However, the taxonomy allows us to see the variations in the MSR approaches with regards to the underlying methodology. For example, approaches using metadata analysis work mostly at the physical-level granularity of artifacts in the software repositories, whereas, the source code differencing approaches work at the logical-level and fine-grain granularity of source code entities. Clearly, the layered taxonomy is effective in drawing similarities and variations in the context of MSR. From the three tables it is clear that all the approaches have corresponding components in all four layers of the taxonomy, thus showing the expressive quality of the taxonomy.
Table 4 we present the approaches alternatively organized by what evolutionary task they are addressing or studying. We’ve grouped them into ten relatively broad categories. For example, all the approaches that investigate “the classification of changes to a system” appear under the “Change Classification/Representation” category. This table not only presents a task-oriented view of the surveyed literature but also a good feeling of the breadth of the tasks addressed by MSR approaches and the amount of research effort that has been put forth towards each. Furthermore, it also shows that some approaches address multiple tasks. For examples, Görg and Weiβgerber [Görg, Weiβgerber 2005a] detect refactorings and also provide visualization for their comprehension, and Livshit and Zimmermann [Livshits, Zimmermann 2005] mined call patterns and also used them for detecting bugs. The following section now contains the details of the literature survey from which our taxonomy is derived.

2.7 MSR – A Comprehensive Survey

We now survey a large number of methodologies that have been invented or adopted for the purposes of MSR. The section is organized by the adopted methodology (subsections) and the purpose of MSR (sub-subsections.) The presentation of the subsections is ordered from the traditional software engineering methodologies (e.g., source code static analysis) to the adopted techniques from other domains (e.g., data mining.) Within each methodology, the individual MSR approaches are discussed with regards to the investigated research questions or interests (i.e., the purpose.) The evaluation of a MSR investigation is discussed with regards to the subject software
systems, the analyzed history, assessment measures, and the results. In cases where multiple papers are available on a particular MSR approach, the survey tends to bias towards the detail or exclusive discussion of the (most recent) paper providing the most comprehensive information on the largest combination of the considered dimensions. This organization gives perspective on the spectrum of techniques and the various purposes for which they are utilized. Furthermore, it helps to assess the quality of a particular technique in the context of the purpose(s) from the results of the conducted evaluations.

2.7.1 Metadata Analysis

A number of methodologies have been proposed for a variety of purposes that utilize the metadata stored in software repositories. The metadata used ranges from what is typically found in the open-source software repositories (e.g., CVS or Bugzilla) to what exists in very sophisticated (i.e., highly customized) configuration-management systems used in industry. Examining metadata is a straightforward first choice as it is readily accessible (e.g., cvs log command or snapshots of a bug database.) In addition, considering only metadata avoids any issues with extracting facts from, and processing, the actual source code and difference data (e.g., parsing non- compilable and incomplete source code.) Sophisticated software repositories, particularly used in the development of industrial products, can additionally form explicit links among the metadata (e.g., CVS deltas corresponding to a particular bug.) Therefore, MSR approaches based only on metadata have been studied extensively for multiple purposes and arguably form the largest portion of the past and current MSR efforts. In the rest of this subsection, we
discuss many such approaches that have employed lightweight methodologies to analyze metadata, such as regular expressions, heuristics, and common-subsequence matching. Our discussion is organized with regards to the purpose.

*Logical Couplings and Change Patterns*

In the work presented by Gall et al. [Gall, Hajek, Jazayeri 1998; Gall, Jazayeri, Krajewski 2003], common semantic (logical and hidden) dependencies between classes due to addition or modification of a particular class are detected, based on the version history of the source code. This work seeks answers to the following representative questions:

- Which classes change together?
- How many times was a particular class changed?
- How many class changes occurred in a subsystem (files in a particular directory)?
- How many class changes occurred across subsystems?

A sequence of release numbers for each changed class is recorded (e.g., class $A = <1, 3, 7, 9>$.). The classes that changed in the same release are compared in order to identify common change patterns based on the author name and time stamp from *CVS* annotations. Classes that changed within the same time stamp (in a four minute window) and author name are inferred to have dependencies.

This technique is applied on 28 releases of an industrial system written in Java with the cumulative size of 500 KLOC. The authors reported that logical couplings were revealed with a “reasonable” recall when verified manually with the subsequent release.
The authors suggest that logical coupling could be strengthened by additional information such as the number of lines changed and the CVS comments.

The analysis was further extended to include metadata from other types of software repositories. The contents of the CVS log files and bug reports from Bugzilla are integrated in a SQL database [Fischer, Pinzger, Gall 2003]. This data is used to trace the origin and modifications of files in the evolution of a system. A heuristic-based merge-point identification algorithm is presented for including the evolution of files along the branches. Further, heuristics using regular expressions are used to map the CVS deltas (commit messages) to the bug reports in Bugzilla. A study was conducted on the history of Mozilla project as found on December 14th 2002. The project history consisted of 433,833 modification reports (CVS deltas), out of which 23,540 were identified as linked to bug reports in Bugzilla. Overall, 158,491 references to bug reports were found (including the above 23,540 modification reports.) Furthermore, the modification history of each file in the project was determined on the scale of release numbers (CVS symbolic tags stored in the log of each file and extracted here with regular-expression matching.) A sequence of release numbers is listed for each file in which it is modified including the one in which it was added. Moreover, the number of different categories of bug reports (e.g., normal, blocker, etc) is associated with each file. The system-level evolution of Mozilla is reported. In total 56 releases were shown to have approximate linear growth per release. In the last 14 releases (i.e, last quarter between the releases 43 and 56), 50% of the files changed (half of which were added.) Also, logical couplings between files were determined based on the link of a file with the bug reports in the Bugzilla and
including all the other files that also referenced the same bug reports. In a reported example, a particular file linked with 33 bug reports was found to be logically coupled with 456 other files.

The above technique was also used to identify change dependencies in the source code across multiple products of a product family [Fischer et al. 2005]. A case study is reported on the CVS repositories of FreeBSD, NetBSD, and OpenBSD variants of BSD Unix where the number of common files between a pair of these products ranged from about 3,800 to 7,000. In order to establish change dependencies, the CVS log records of each project were analyzed to determine the presence of other projects keywords (e.g., A FreeBSD CVS log records were analyzed for keywords netbsd, openbsd, and linux.) The distribution of this information was studied from a period from 1994-2004. The results indicate that OpenBSD continues to be more decoupled from the rest of the projects.

*Heuristics for Change Predictions*

Hassan and Holt [Hassan, Holt 2004] use a variety of heuristics, such as developer-based, history-based, call/use/define relation, and code-layout-based (file-based) to predict the entities that are candidates for a change on account of a given entity being changed. CVS metadata is lexically analyzed to derive the set of changed entities from the source code repositories. The following assumptions were used: changes in one record are considered related; changes are symmetric; and the order of modification of entities in a change-set is unimportant. The authors briefly state that they have developed techniques to map line-based changes to syntactic entities such as functions and variables, but it was not completely clear the extent to which this is automated.
These heuristics are applied to five open-source projects written in C. General maintenance records (e.g., copyright changes, pretty printing, etc) and records that add new entities are discarded. The best average precision and recall reported in [Hassan, Holt 2004] (specifically the author’s Table 3) was 12% (file-based) and 87% (history) respectively. The call/use/define heuristics gave a 2% and 42% value for precision and recall respectively while the hybrid heuristics did better.

**Bug-Fixing Change Analysis**

A combination of information in the CVS log file (change deltas) and Bugzilla is used to study fix-inducing changes, i.e., that introduced new changes to fix an earlier reported problem, by Sliwerski et al. [Sliwerski, Zimmermann, Zeller 2005]. The deltas in the CVS log file are grouped into transactions by using the sliding-window approach. Regular-expression matching on the commit messages and text descriptions in Bugzilla along with heuristics are used to determine the CVS deltas that are related to a change that fixes a bug. Given such a change (fix), the modified lines are identified by using the cvs diff command. The latest deltas that also affect the involved lines are found using the cvs annotate command. These deltas are further analyzed by heuristics to filter out false positives. The remaining “true” deltas are considered to be the changes that induced a given fix.

The main question investigated is which change properties (such as changes on a specific day or by a certain group of developers) may lead to problems (i.e., more changes)? The approach is validated on two open-source projects, Eclipse (78,954 transactions) and Mozilla (109,658 transactions) as of January 2005. The average size of
transactions which are fixes and lead to further fixes is 3.82. Overall, the fix-inducing transactions are about three times larger than the non-fix inducing transactions. High risk of introducing fix-inducing changes was found on Saturdays and Fridays for Eclipse and Mozilla respectively.

**Characteristics of Different Types of Changes**

German [German 2004c] examined software repositories (CVS) to study the evolution of the email client, *Evolution* between 1998 and 2003. The CVS annotations are used to group subsequent changes into what is termed a *modification request (MR).* The study was directed at characteristics such as the growth in the size of the software, number of files and their type (e.g., source code and configurations) distribution, number of (types of) MRs per month (e.g., MRs involving source code only), most changed files, most active contributors, and contribution/changes in modules. In another study on the same system [German 2004a] the focus is on studying different types of MRs. Following are a set of representative questions that are examined,

- Do MRs adding new functionality differ from MRs fixing bugs?
- Are MRs different in different stages of evolution?
- Do files tend to be modified by the same developer?

The analysis of all the MRs in the history of *Evolution* found that on average the MRs changing source code (codeMRs) consisted of more files than the MRs consisting of bug fixes (bugMRs). The MRs that consisted of only changes in the comments (commentMRs) on average consisted of more files than any other type of MRs. The number of functions added or deleted with bugMRs was overall less than any other type
of MRs. The analysis of 3,094 MRs from the year 2002 found 2,261 codeMRs, 155 bugMRs, and 93 commentMRs. The months October 2002 and November 2002 were identified as a maintenance (bug fixing) period and an improvement (new functionality) period respectively based on a stable release of Evolution. The maintenance period had fewer MRs than the improvement period. Not a single MR in the maintenance period included files from two different modules. Also, in the improvement period MRs that included files from different modules were restricted to two specific modules and three specific files. Most files were modified multiple times by the same developer. There were a few cases where multiple developers modified a set of common files however those files belong to the same module.

**Formalism for Querying Metadata**

Metadata, such as those found in the CVS log files, are modeled using a graph representation by Hindle and German [Hindle, German 2005]. A query language based on first-order and temporal logic, namely SCQL, is defined to facilitate questions on changes at a level expressed by the metadata. The approach is demonstrated via three example queries expressing the following questions:

- Does an author exist whose only modifications were to files already modified by another author?
- What is the proportion of MRs containing a unique set of files that are never involved in any other MR?
- Does an author exist whose changes are bounded within a single directory?
The above queries were evaluated on the five open-source systems *Evolution, Gnumeric, OpenSSL, Samba, and modperl*. The CVS repositories contained between 300 to 4,748 files, and between 1,398 to 18,573 versions. The results were that three systems had an author whose only modifications were to files already modified by another author, four systems had an author whose changes were bounded within a single directory, and the proportion of MRs that contained a unique set of files that were never involved in another MR was between 0.002 and 0.015.

*Characteristics of Small Changes*

The focus of a study presented by Purushothaman and Perry [Purushothaman, Perry 2004; 2005] is to understand the impact of small changes, particularly one-line changes, with regards to faults, relationship between different types of changes (i.e., add, delete, and modify), reason for the change (i.e., corrective, adaptive, and perfective), and dependencies between changes. A change is considered to be a one-line change if there was at least one modification to a single line, at least one line was replaced by a single line (i.e., multiple lines deleted followed by an addition of a single line), a new statement was added between existing lines, or a single line was deleted.

The research questions addressed are restated as follows:

- How do small changes differ from other changes?
- What is the relationship of the types and purposes of changes over time?
- What is the relationship between the size of a change and its type and purpose?
- What effect does the size, type, and purpose of a change have on the likelihood of producing a fault?
These questions are evaluated by an empirical study on the first 15 years of the history of 5ESS, a telephone-switching subsystem. This software was developed in a very well-defined development environment including a sophisticated change tracking system by a group of well-trained and qualified developers. A change is tracked from a domain-level description (in the form of an Initial Modification Request, IMR which is a textual description of a feature request) to a set of logical units (in the form of a Modification Request, MR which is a concise assignment to a single developer) to a set of physical units (i.e., files and lines.) Heuristics developed by Mockus and Votta [Mockus, Votta 2000] are used to classify each change as corrective, perfective, adaptive, or inspection. The results of this study are summarized below,

- Approximately 10% of changes were one-line changes.
- About 50% of changes involved at most 10 LOC, and about 95% of changes involved at most 50 LOC.
- The perfective category consisted of approximately 2.5% one-line additions and approximately 10% of other types of one-line changes.
- Most changes were found to be adaptive and contained addition of code.
- About 40% of changes introduced for fixing defects introduced at least one more defect.
- Only 4% of the one-line changes caused a defect.
- The chances of a one-line addition and modification causing a defect are approximately 2% and 5% respectively.
- The chance of a defect occurring for a change that involved more than 500 LOC is
about 50%.

• It remained inconclusive whether deletions of less than 10 LOC cause a defect.

**Searching and Browsing Source Code**

A web-based, source code search tool, namely *CVSSearch*, built on top of the commands *cvs log* and *cvs diff* and utilizing the text in the *CVS* commit messages is presented by Chen et al. [Chen et al. 2001]. A historical context for all the lines in the latest version of a system is formed by associating each line with all the commit messages in a software repository. The tool accounts for the addition and deletion of lines, and uses a string-alignment algorithm for more precise modifications than provided by *diff*. A user query is specified in terms of keywords (e.g., *login*.) The tool displays all the files that have lines matching at least one of the keywords with a link to the matched lines. Furthermore, *CVSSearch* also executes the same query with *grep* and reports the matching files also with a link to the matched lines.

The authors report the evaluation of the *CVSSearch* tool as applied to five KDE applications. These applications range between 24 KLOC and 49 KLOC approximately and average between 10.8 and 38.3 revisions/file. Seventy-four students who were unfamiliar with the source code of the considered applications were selected for the study. *CVS* comments performed better on 40%, *grep* performed better on 32%, and both performed equally on 28% of the 703 tested queries. Deciding whether *CVS* comments are better than *grep* or vice versa with p-values, overall *CVS* comments did better than *grep* for all the applications. However, the results are inconclusive for individual applications.
Successful Open source Development

The work by Dinh-Trong and Bieman [Dinh-Trong, Bieman 2005] is an external validation of five of the seven hypotheses pertaining to successful open-source software development given by Mockus et al. [Mockus, Fielding, Herbsleb 2002] from their empirical studies on Apache (developed without major commercial support and managed by a voluntary organization) and Mozilla (developed with major commercial support and managed by a profit organization.) This work is an extension of the authors’ prior empirical study on the nine-year history of the FreeBSD project. The principle objective is to determine whether the hypotheses developed in [Mockus, Fielding, Herbsleb 2002] represent general trends for successful open-source development. Here, only five hypotheses are examined with two left out due to inapplicability to the FreeBSD project and a lack of data for validation. Additional goals were to find the common characteristics of the processes used in successful open-source development and the quality of the resulting software. The research questions of interest are directly stated below,

- “What were the processes used to develop Apache and Mozilla?”
- “How many people wrote code for new functionality? How many people reported problems? How many people repaired defects?”
- “Were these functions carried out by distinct groups of people, that is, did people primarily assume a single role? Did large numbers of people participate somewhat equally in these activities, or did a small number of people do most of the work?”
- “Where did the code contributors work in the code? Was strict code ownership
enforced on a file or module level?”

• “What is the defect density of Apache and Mozilla code?”

• “How long did it take to resolve problems? Were high-priority problems resolved faster than low-priority problems? Has the resolution interval decreased over time?”

The facts and data to answer the questions in the context of the open-source projects were collected from email archives (sent to freebsd-bugs@FreeBSD.ORG), a bug database (GNATS), and a CVS repository (log file.) Only problem reports related to the source code correction (i.e., classified as sw-bug in the GNATS database) that were present in both the stable and current branches were considered. The authors developed tools to process the log records in the CVS repository to determine the number of contributors, the number of changes committed by each contributor and the aggregate number of changes, and the total number of lines added. The CVS deltas that contain the keyword PR (Problem Report) in the commit message were regarded as updates to fix problems whereas others were attributed to new features. A distinction was made between source code files (.h and .c) and other files (readme, makefile.) The name of the person who reported the PR to the email list was extracted from a line starting with the keyword “Originator.” Also, statistics such as the number of people who reported the bugs and the number of bugs reported by each person were obtained. The same data from the four commercial systems used by Mockus et al. [Mockus, Fielding, Herbsleb 2002] were also used in this study. The results of this study show support for two
hypotheses and suggest revision of the remaining three. The Modified Hypotheses are directly stated below,

“H1` A core of 15 or fewer developers will control the code base and contribute most of the new functionality. A group of 50 or fewer top developers at any one time will contribute 80 percent of the new functionality. The group will represent less than 25 percent of the set of all developers.”

“H2` As the number of developers needed to contribute 80% of open source code increases, a more well-defined mechanism must be used to coordinate project work.”

“H3` Defect density in open source code releases will be lower than commercial code that has only been feature-tested. If an open source system has a mechanism to separate unstable code from stable code or “official” releases, then the defect density of the stable code releases will be equivalent to that of commercial code after release.”

Developer Identities

It has been observed [Koch, Schneider 2002; Mockus, Fielding, Herbsleb 2002; Robles, Koch, González-Barahona 2004] that source code contributions to open-source development follow a Pareto distribution i.e., a small number of participants (i.e., 20%) contribute a bulk of the project (i.e., 80%). One explanation for this distribution is that the same developers, with possibly different identities, contribute to various repositories (e.g., user-id for CVS repository, developer’s name in source code, email address in the project mailing lists and bug-tracking system.) Robles and Gonzalez-Barahona [Robles, González-Barahona 2005] present a methodology based on heuristics such as spatial locality (e.g., in source-header comments an email address and a developer’s name occur
together) on the identities collected from various repositories (e.g., email archives, CVS repositories, Bugzilla.) The approach is validated on the GNOME project. The examined data set consisted of 464,953 email messages from 36,399 distinct email addresses, 123,739 bug reports from 41,835 reporters and 382,271 comments from 10,257 posters, and approximately 2,000,000 CVS commits by 1,067 committers. The results indicate that these identities actually correspond to 34,648 unique persons. The authors further plan to investigate gender and nationality distribution.

Relationships between Bugs/Features

An examination of the bug reports (BRs) from a bug-tracking system is discussed with regards to various formal (i.e., explicitly represented and stored) and informal (i.e., derived from the content and not explicitly stored) relationships between them in Sandusky et al. [Sandusky, Gasser, Ripoche 2004]. A bug-tracking system typically contains a category field that explicitly stores the relationship (if any) of a bug report with regards to others. These relationships are a result of bug duplication and dependency. Duplications result from the multiple reporting of the same bug. Dependencies arise in situations such as when a bug fix cannot be performed until another bug is resolved, or a bug fix blocks resolution of other bug fixes. Moreover, bug reports are linked by informal relationships that create semantic associations between them. Such relations are typically derived from the description of, and/or comments posted for, a bug report (e.g., texts such “also refer to Y” and “See the fix of Y...”).) Taken as a whole, these relationships create bug-report networks (BRNs.) Such networks help reduce duplication of effort in solving the same problem, support a bug fix by pointing to
other similar solutions, or help in the identification of critical bugs. However, almost all the relationships are still manually discovered and maintained. A random set of 385 bug reports was selected from a population of approximately 182,000 bug reports that were opened over a period of five years in an unspecified open-source project. Almost 65% of them had either a formal or informal relationship with at least one other bug report. Duplications accounted for 43% and dependencies accounted for 19%, with 33% attributed to informal relationships.

*Software Defects/Faults and Predictors*

Ostrand and Weyuker [Ostrand, Weyuker 2004] use the data from a bug-tracking system to construct a fault-prediction tool based on a statistical model (i.e., a negative binomial regression model.) Metrics for a file such as LOC, age in versions, number of faults in a previous version, and source code language are considered as independent variables. The identification of MRs that represents faults/defects are performed by examining the roles of, or interviewing, the reporters. The MRs reported by testers are considered as strong candidates for faults, whereas those reported by developers require further inspection. The goal of this tool is to enable testers to obtain an ordered list of fault-prone files in the next release. The testers can query the tool for a set of files based on the percentage of the project or percentage of faults. For example, a list of the 20% of the files in the next release that are predicted to have the most faults; or a minimum set of files that are predicted to contain at least 5% of the faults. In one of the studies reported by the authors on a large AT&T project with 17 successive releases, the top 20% of the files predicted by the model were found to contain between 71% and 92% of the actual
faults with an average of 83%. A query for the minimal set of files containing at least 80% of the faults produced less than 20% of the files in some releases.

Evolution of a Software Distribution

Robles et al. [Robles et al. 2006] study the evolution of a software distribution. A software distribution refers to several software applications/libraries (typically independently developed) that are distributed as a single integrated system. Their interest is to study the characteristics of the number of packages, lines of code, use of programming languages, and sizes of packages/files with regards to the evolution (i.e., multiple versions) of a software distribution. Robles et al. work differs from previous research that investigated only a single version of a software distribution. Five stable releases of Debian (a Linux-based distribution) within seven-year duration between the 2.0 release and the 3.1 release are examined. The file Sources.gz (available in each release of Debian) consists of information such as names, binaries/source files, version, and maintainers of the included packages (i.e., applications/libraries) in a release. The tool SLOCCount (http://www.dwheeler.com/sloccount/) is used to process this file and help compute the above measures. This study reported the following observations,

- The overall size of Debian (order of MLOC) approximately doubled every two years.
- There are relatively fewer large packages (over 100 KLOC) than small packages (1 KLOC to 50 KLOC) in all the releases.
- The large packages were shown to increase in subsequent releases. However, more small packages were added.
• There was not a substantial difference in the mean package sizes across the releases (around 23 KLOC.) The above two observations were given as one possible reason for this observation.

• About 15% of the packages remained unchanged since the release 2.0.

• The most used programming language in each release is C. However, the relative percentage of C is decreasing in subsequent releases (from 76.7% in release 2.0 to 55.8%). The usage percentage of the interpreted languages such as Python and Perl shows a sharp growth.

• The file sizes of programs written in the procedural and structural languages are larger than those written in the object-oriented languages.

Completeness of ChangeLog Files

Chen et al. [Chen et al. 2004] examines the viability of using the change information between two successive releases typically recorded in a single file, namely ChangeLog, for research investigations. The specific question of interest is does ChangeLog record the complete set of source code changes performed. The cross-referencing tool lxr is used to compute the source code differences between two versions. These source code differences are then compared with the entries in the ChangeLog. Furthermore, each change was manually categorized as a corrective, enhancement, code rearrangement, or a comment change. The ChangeLog files in at least three releases of the open-source software GNUJSP, GCC-g++, and Jikes are used to evaluate the research question. The changes excluded in the ChangeLog files ranged between 3.7% and 78% with an average of 22.2%. With regards to individual systems, the analysis of
four releases of GNUJSP showed (weighted averages) 31.3% overall, 52.9% corrective, 9.7% enhancement, 55.5% rearrangement, and 60% comment changes were not found in the ChangeLog files. The analysis of three releases of GCC-g++ showed (weighted averages) 10.8% overall, 7.5% corrective, 7.4% enhancement, 0.0% rearrangement, and 44.4% comment changes were not found in the ChangeLog files. The analysis of three releases of Jike showed (weighted averages) 24.5% overall, 15.3% corrective, 7.8% enhancement, 91.7% rearrangement, and 46.0% comment changes were not found in the ChangeLog files. The authors note that incompleteness and inaccuracies of ChangeLog should be carefully considered when using them as a basis for research investigations.

2.7.2 Static Source Code Analysis

Source code is one of the most important artifacts available from source code repositories as its evolution largely contributes to the overall software evolution. Generally, source code repositories provide the capability to access source code at any stage (i.e., version) in the history of the software evolution. This allows us to study software evolution not only from release-to-release but examining changes in individual versions.

A number of MSR approaches use static program analysis to extract facts and other information from versions of a system. These approaches span across a wide range of available techniques for parsing, processing, and extracting facts from source code. This information is used to compare the different versions. In this section, we discuss how static analysis has been used in the context of MSR. Again our discussion is organized with regards to the purpose of MSR.
Bugs Finding and Fixing

In an approach presented by Williams and Hollingsworth [Williams, Hollingsworth 2004; 2005a], bug-fix information is automatically mined from the source code repository to improve bug finding/fixing tools. The type of bug considered is a function-return-value check. The existence of this type of bug in the considered systems (Apache and Wine) is determined by manual inspection of the source code repository. A custom tool for detecting function-return-value checks is developed based on a traditional compiler-like parser. The tool combs through all the changed files across versions and identifies a list of functions that are considered to be function-return-value bug fixes. Such a bug is considered fixed in the historical context if a conditional statement in a subsequent version that was not present in the preceding version guards a further use of a return value.

The bug finding is based on both the historical context (data mined from the source code repository) and contemporary context (current version.) If a change involves a call to a function present in the list of functions obtained from the history and the return value is used before being checked, it is flagged as a warning (potential bug.) Furthermore, if a return value check after a function call appears in more than 50% of the instances in the current version, the other call sites without a return-value check are flagged as warnings. The description of the warnings includes the physical attributes (file name, line number) of the involved call sites. The warning candidates are presented in order from the most likely to least likely and divided into two halves. The warnings
derived from historical context (a.k.a. History-Aware ranking) are given a higher priority than those obtained by the contemporary (a.k.a. Naive ranking) context alone.

The proposed methodology was evaluated on two software projects: Apache and Wine. The effectiveness of bug fixing and the ranking were the main assessment factors. The false-positives rate in both the cases was reported lower when historical context (Apache 0.61, Wine 0.65) was considered versus the contemporary context alone (Apache 0.75, Wine 0.82.) Also, the sets of warnings found in both the contexts were proper intersecting sets (i.e., there exists items reported by one and not reported by the other.) The evaluation of the ranking indicates there were instances of Naive ranking outperforming HistoryAware ranking in terms of precision. However, overall there was a better precision for the HistoryAware ranking for the 50 top-ranked warnings. The effectiveness of the mined information is also evident from the results of recall. Since most of the "true" bugs are already identified (with better precision) in the warnings presented in the upper half (HistoryAware ranking), the recall for the warnings in the lower half (Naive ranking) is also improved.

Factors for Successful Software Reuse

The study presented by Selby [Selby 2005] is an investigation of the factors that characterize successful software reuse in large-scale systems. Both design and implementation factors characterizing successful software reuse are examined by an empirical study on the repositories of 25 systems written in Fortran ranging from 300 to 112,000 LOC, developed by NASA in a highly reuse-based environment (i.e., 32% reuse per project.) The study is conducted and evaluated based on the goal-question-metric
(GQM) paradigm. The goals are set, questions are devised to fulfill each goal, and metrics are defined to answer questions. The classification of the size of the project (i.e., small and large) and the classification of the modules based on the type of reuse (without, slight, major and new) were based on the statistical analysis in a nonparametric ANOVA (analysis-of-variance) model. The data was collected from forms manually entered by the developers (maintainers), and static analysis of the source code repository, both collectively stored in a relational database.

The modules reused without, with slight, and with major revisions were found to be 17.1%, 10.3%, and 4.6% respectively. There was no substantial difference between the modules reused without and slight revisions between small and large projects. Large projects had more modules reused with major revisions than small projects. Higher amount of reuse lowered the development efforts. The number of interfaces in modules decreased in the order of: major revisions, slight revisions, and without revisions. The module size decreased in the order of: major revisions, new, slight revisions, and without revisions. Overall, the module reused without revisions had better documentation. The faults per source line and the fault-correction effort were the lowest in modules without revisions and the highest in modules with major revisions, whereas the changes per source line and the change correction efforts showed the opposite.

Function Usage Patterns

A method to automatically detect function usage patterns is presented by Williams and Hollingsworth [Williams, Hollingsworth 2005b]. Specifically considered are the patterns called after (i.e., a function $B$ is called after function $A$) and conditionally
called after (i.e., a function $B$ is called after function $A$ but is guarded by a condition.)

Mining the source code repository identifies the instances of such usage patterns. The goal is to find new instances in the current version. A C parser is used to identify function calls. Instances are additionally categorized into groups, e.g., debug and string manipulation. A pair of function calls found within a distance specified by the number of lines of code is considered to follow the function usage pattern. The tool is applied to the software repository of the Wine project. Overall, about 50 million instances were reported with 2,175 and 65 instances of patterns identified as new at least 10 and 100 times respectively. For the results by individual groups, we refer the readers to Tables 1 and 2 in [Williams, Hollingsworth 2005b].

Incomplete Refactorings

A method for identifying incomplete refactorings including Add/Remove Parameter and Rename Method across super-classes, sub-classes, and sibling-classes for Java programs is described by Görg and Weiβgerber [Görg, Weiβgerber 2005b]. Such incomplete refactorings may cause errors (change in behavior) that are typically not captured by a compiler (e.g., a method is inherited rather than being overwritten.) This approach is capable of handling refactorings that take more than one version to complete due to the practice of small incremental change.

The detection of the considered refactorings starts with the first modified version of a file in the CVS repository. A lightweight parser is used to obtain all the classes and methods from this file and its immediate next version. On comparing the two versions, lists of added, deleted, and common methods between these two versions are obtained.
Further analysis of these lists results in the identification of Add/Remove Parameter and Rename Method refactorings between two versions of a class. However, these results may be incomplete, requiring an examination of classes in the inheritance hierarchy. Such classes are further analyzed to determine inconsistencies indicative of an incomplete refactoring. For example, a Rename Method refactoring was applied to a base class, but not to the corresponding method in the derived class. Notice that this may change on further analyses of the next versions, if the found inconsistency is resolved.

A preliminary case study on two open-source projects, jEdit and Tomcat is described. The approach reported five (two methods in sub-classes, and three methods in sibling-classes) and seven (three methods in sub-class and four methods in sibling-classes) incomplete-refactoring candidates for jEdit and Tomcat respectively. Except for the two methods in the sub-classes with jEdit, none of the methods were found to be completely refactored in the later versions.

**Function-Interface Changes**

A fine-grain analysis and classification of function-signature changes is presented in Kim et al. [Kim, Whitehead, Bevan 2005]. A fact-extraction tool developed by the authors is used to identify function signatures present in all versions. The obtained function signatures are processed to determine the exact changes across versions and classify them with the help of a semi-automatic tool also developed by the authors. Three broad categories are defined based on the impact on the data flow between a called function and a calling function, namely *data-flow invariant*, *data-flow increasing*, and
data-flow decreasing. These categories are further divided based on the exact changes in the function signature (i.e., function name, parameters, and return type.)

The fine-grain analysis and classification of function signatures was used to investigate the following research questions:

• How frequent are function-signature changes?
• What are the commonly occurring types of function-signature changes?
• What is the frequency distribution of each type?
• Do function signatures have a common pattern of evolution?

Eight open-source projects written in C (we refer the readers to Table 1 in [Kim, Whitehead, Bevan 2005] for further details) were analyzed with regards to the above questions. The distribution of the number of functions with regards to their number of signature changes, varying from 0 to 16, is presented. The authors report that in case of the Subversion project, 77% of functions were never involved in a signature change, and 95% of the function involved in a signature change had fewer than three signature changes. The most commonly-occurring changes were parameter addition (52.13%), complex type changes (30.5%), and parameter deletion (22.75%), whereas the least commonly-occurring changes included array/pointer and primitive-type changes. Furthermore, it was found that a function signature might follow a particular sequence of changes in successive versions (e.g., a parameter addition followed by a parameter deletion.) A modified version of the longest common subsequence algorithm (LCS) was used to detect commonly occurring change patterns in a function signature. The authors foresee the application of this in predicting future changes in a function signature.
2.7.3 Communication Via Source Code Comments

Ying et al. [Ying, Wright, Abrams 2005] presents an interesting use of mining the source code comments developed in the Eclipse environment. The Eclipse environment supports a task-specific description in the source code comments (e.g., “Mike, please fix this...”) via the task-tag mechanism (e.g., “TODO” tag.) Such comments are termed *task comments*. The task comments form an additional source that captures the communication about changes that are, or were planned to be, performed.

A study on the CVS repository of the proprietary AWB project written in Java as of February 9, 2005 is described. It consists of 2,213 files that were found to contain 221 task comments (i.e., comments with a string “TODO”). The task comments were analyzed for their content and their intended purpose. It was found that they are used for point-to-point and group communication, pointers to change request in the change/bug/defect tracking system, bookmarks on past tasks that may need further work, current and future tasks, location markers, and concern tags for marking distributed places in the code that need a similar change. The content analysis shows that a task comment may include an author’s identity and change-request identifiers that may be useful for MSR applications.

2.7.4 Source Code Differencing and Analysis

Source code repositories contain differences between versions of source code (i.e., difference data.) As discussed in Section 2.5.1, this difference data is file and line based. To further extend MSR with regards to changes, researchers have proposed methods to derive and express changes from source code repositories in a more source
code “aware” manner (i.e., syntax and semantic.) To help support this view of MSR, information from source code or source code models is utilized. Here, we discuss these MSR techniques in light of how changes are expressed and the MSR questions asked about the changes. Our discussion is organized with regards to the purpose of MSR (i.e., level-two subsections), which in this case is the different ways of expressing source code differences.

Semantic Differencing

The tool *Dex* is presented by Raghavan et al. [Raghavan et al. 2004] for detecting syntactic and semantic changes from a version history of C code. All the changes in a patch are considered to be part of a single higher-level change, e.g., bug fix. Each version is converted to an abstract semantic graph (ASG) representation. A top-down or bottom-up heuristics-based differencing algorithm is applied to each pair of in-memory ASGs. The differencing algorithm produces an edit script describing the nodes that are added, deleted, modified, or moved in order to achieve one ASG from another. The edit scripts produced for each pair of ASGs are analyzed to answer questions from entity-level changes such as how many functions and function calls are inserted, added or modified, to specific changes such as how many *if* statement conditions are changed. *Dex* supports 398 such statistics. This technique was applied to version histories of *GCC* and *Apache*. Only bug-fix patches were considered (deduced from the *CVS* metadata), 71 for *GCC* and 39 for *Apache* respectively. The differencing algorithm takes polynomial time to the number of nodes. Average time of 60 seconds and 5 minutes per file were reported for *Apache* and *GCC* respectively on a 1.8 GHz Pentium IV Xeon 1GB RAM
machine. The six frequently occurring bug-fix changes as a percentage of patches in which they appear are reported. *Dex* reported 378 out of 398 statistics were always computed correctly. An average rate of 1.1 incorrect statistics per patch was reported.

In another approach Neamtiu et al. [Neamtiu, Foster, Hicks 2005] use a partial AST matching algorithm for detecting semantic changes (i.e., additions, deletions, and modifications) between a pair of versions of a C program. Here, global variables, types, and functions are the entities of interest. The differencing is actually a two-step process: AST matching and change detection. The AST matching algorithm takes ASTs of two functions and matches the type and name of all the local and global variables within their bodies creating a bijection mapping between matched entities. The matching algorithm terminates on detection of the first mismatch and therefore may fail to identify the matching pairs in the remainder of the tree. Also, functions that are renamed are never matched giving another source for missing matching pairs. As a result, some entities may be identified as added/deleted instead of actually being renamed.

For detecting changes between a pair of files, if a function with the same name occurs in both files, it is considered to be modified semantically only if there are changes in the body other than the renaming pairs identified by the bijections. Functions with different names are identified as added/deleted. Similarly, variable names and types are reported to be added, deleted, or renamed with an additional requirement for a strict structural isomorphism check for type equality.
The primary focus was to support the dynamic software updating (DSU) technique (changing software without halting its execution.) The authors pose three questions for a primarily evaluation to help achieve the above goal:

- Are functions and variables deleted frequently relative to the size of the program?
- Do function prototypes change frequently?
- Are changes to type definitions simple?

Three tools, *Vsftp*, *Apache*, and *OpenSSH* were selected to investigate the above questions. For *Vsftp* and *Apache* almost no functions were deleted and the size of the functions remained almost constant. However, in *OpenSSH* functions were deleted at a steady rate. The function prototypes changed less frequently in *Vsftp* and *Apache* than in *OpenSSH*. Most type-definition changes involved only one or two entities with the exception of *OpenSSH* where more than two entities were changed regularly. This implies *Vsftp* and *Apache* are more suitable for dynamic software updating while *OpenSSH* involves a risk.

*Syntactic Differencing for Fine-grain Analyses*

Maletic and Collard [Maletic, Collard 2004] present a syntactic-differencing approach called *meta-differencing* which answers syntax-specific questions about differences. This is supported by first encoding the AST information directly into the source code via an XML format, namely *srcML*, and then marking added, deleted, or modified sections in an extended srcML format, namely *srcDiff*. The types and prevalence of syntactic changes are then easily computed. Queries are performed as XPath expressions on the *srcDiff* format supporting questions such as:
- Are new methods added to an existing class?
- Are there changes to pre-processor directives?
- Was the condition in an if-statement modified?

While no extensive MSR case study has been carried out using meta-differencing, it does support the functionality necessary to address a range of these problems. Additionally, the method is fairly efficient and usable with run times for translation similar to that of compiling and computation of the meta-difference around five times that of a textual diff.

Identification of Refactorings in Changes

Weißgerber and Diehl [Weißgerber, Diehl 2006] present a technique for identifying changes that are refactorings. The line-based differences of files in a CVS commit are mapped to the differences in syntactic entities (e.g., class and method names.) The type of changes (e.g., add, delete, and modify) in the syntactic entities are then analyzed to infer the refactorings Move/Rename Class/Interface, Move Field, Move Method, Rename Method, Hide/Unhide Method, and Add/Remove Parameter. Three open-source systems ArgoUML, jEdit, and Junit are used to examine whether refactorings caused less bugs than other changes. A change is considered to introduce a bug if a bug report is opened in a certain number of days after that change. Metrics, including the number of changed entities, number of bugs per changed entity, and the number of refactorings per changed entity, are used to indirectly correlate with the number of bugs per refactoring. The number of versions considered for ArgoUML, jEdit, and Junit were 65,593, 10,726, and 1,707 respectively. It was found that a high ratio of
refactorings per days followed by no increase in the ratio of bugs per days was prevalent in most periods of history, an indication that refactorings are less bug prone. However, they also found instances where a high ratio of refactorings to days was followed by an increase in the ratio of bugs to days.

Dig et al. [Dig et al. 2006] used a combination of syntactic and semantic analyses to uncover refactoring changes that occur between two versions of a system. In this approach, syntactic analysis is first performed via lightweight AST to identify source code entities along with their fully qualified names. Then an information retrieval technique, namely Shingles encoding, is used to identify pairs of source code entities that are refactoring candidates (i.e., old version before refactoring and new version after refactoring.) The Shingles encoding technique basically finds pairs of source code entities with similar textual contents. In order to further refine the candidate refactorings (i.e., reduce false positives), calls from and to a source code entity with both before and after refactored versions are analyzed. If they continue to have similar calls in both versions, a candidate refactoring is confirmed as a true case. An Eclipse plugin is developed with strategies to detect seven types of refactorings including multiple types performed in a single change (e.g., both of the refactorings method rename and signature change.) Two major versions of three open-source software Eclipse.UI, struts, and jHotDraw are used for evaluation. Refactorings were identified with both precision and recall over 85%. Results of manual examination from previous results were used for validation. The end goal of this work is to support automatic replaying of refactorings performed in a component to the clients of that component. Henkel and Diwan [Henkel,
Diwan 2005] have a similar goal but record refactorings as they are performed by a component/library developer in the Eclipse IDE. This method is realized as a plugin within the Eclipse IDE.

Changes in Micro Patterns

Kim et al. [Kim, Pan, Whitehead Jr. 2006] study the changes in micro patterns. A micro pattern is a programming idiom for a class in Java [Gil, Maman 2005]. The interest is in analyzing changes with respect to the type of a micro pattern of a class (e.g., from a Stateless to a RestrictedCreation type.) Their further goal is to correlate the changes in the micro patterns with the reported bugs. The CVS repositories of three open-source projects ArgoUML (1,262 versions), Columbia (1,652 versions), and jEdit (1,449 versions) are used. The coverage of the different types of micro patterns seen in the considered latest versions of ArgoUML, Columbia, and jEdit are 55%, 79%, and 81% respectively. The changes in the type of a micro pattern in ArgoUML, Columbia, and jEdit are 6%, 5%, and 4.1% respectively of the total micro-pattern changes. The top twenty frequently-changed micro patterns and top twenty bug-prone micro patterns are listed for all three considered projects. There is almost no similarity in the observed micro patterns with regards to both lists across the projects. Two different periods of jEdit were found to exhibit identical bug-prone behavior with changes in micro patterns. The authors note that the correlation between the number of changes in the micro patterns and the introduction of bugs remains inconclusive.
Detecting Similar Java Classes

Sager et al. [Sager et al. 2006] identified similar classes in Java source code using three tree algorithms: bottom-up maximum common subtree isomorphism, top-down maximum common subtree isomorphism, and tree edit distance. The Abstract Syntax Trees (ASTs) are generated from two versions of the source code then are converted to FAMIX [Demeyer, Tichelaar, Steyaert] trees for language-independent representation. The two FAMIX trees corresponding to the two versions of source code are compared with the tree algorithms. The approach is evaluated on the Eclipse compare plugin (versions 3.0 and 3.1) using the tree-edit distance algorithm (proved best of the three algorithms on the specially devised test cases.) The similarity in the classes between the version 3.0 and the version 3.1 are shown in the form of a heatmap, a two-dimensional plot with a box used to show the similarity. For details we refer the readers to Figure 6 in [Sager et al. 2006].

Studying API Changes

Dig and Johnson [Dig, Johnson 2006] study the API changes between two versions of a framework/library (referred here as component.) The interest is in the classification of the API changes as breaking and non-breaking changes. An API change of a component is considered to be a breaking change if its client application (i.e., an application using the API) fails to compile, link, or produces different output behavior after that change is performed, and an API change of a component is considered as non-breaking if the client application continues to be backward compatible (i.e., changes are local to a component.) A combination of change logs, release notes, help documentation,
developer interviews, and manual examination of source code differences are used to identify and classify API changes. Three open-source frameworks *Eclipse framework*, *struts*, and *jHotDraw*, one open-source library *log4j*, and one proprietary framework *mortgage* written in Java are used to conduct the investigation. Two versions of each system are considered. A total of 51, 136, 58, 38, and 11 API changes were found to be breaking changes respectively. Of the breaking changes, refactorings formed 84%, 90%, 94%, 97%, and 81% of the breaking changes respectively. That is, a refactoring (behavior preserving property) was restricted to a framework/library. However, the impacted parts of its client application (using the old API) were not changed accordingly.

2.7.5 Software Metrics

Software metrics are used to quantitatively assess various aspects of software products, projects, and process. These aspects include size, effort, cost, functionality, quality, complexity, efficiency, reliability, and maintainability of a software artifact, system, or the related process. In this section, we discuss how metrics are used in the context of MSR.

Complexity of Different Changes

The work presented by Nikora and Munson [Nikora, Munson 2003] is an examination of sources of variations in the set of software metrics used to measure a system under evolution. Twelve size and control-flow metrics at the function level are used. Principal Components Analysis (PCA) was applied to identify three distinct domains of variations. Each module in a particular build is represented by a fault index
(FI) value. Basically, FI is a weighted sum of the twelve metrics in proportion to the amount of unique variation contributed by that complexity metric.

A case study on the Mission Data System (MDS) system is described. MDS is a system developed by the Jet Propulsion Laboratory (JPL) managed by NASA. The history of MDS consists of 1,500 builds, 65,000 versions, and more than 15,000 functions. The hypothesis here is that not all the changes contribute equally to the overall complexity of the system – changes to comments could be simple while others may have substantial impact on the structure of software modules. The goal is to verify this hypothesis and further investigate whether structural metrics are suitable for predicating the number of faults introduced in the system and what kinds of changes contribute more in inserting faults than others. The information regarding faults was collected from the analysis of 1,400 problem reports.

The results indicate that not all the builds are equivalent. The control structure of the system changes much more rapidly than others, and a substantial amount of changes are attributed to it. There is a fluctuation of change activities in all the domains across the initial few builds. However, the change activities stabilize after a particular build (i.e., build 247 here) when the control-structure domain becomes the dominant factor. The control structure is most closely associated with the cumulative-fault measure. The variation in the number of faults appears to increase directly with the increase in the complexity of the system.
Change-prone Classes and Change-couplings

In another approach by Bieman et al. [Bieman, Andrews, Yang 2003], a metrics-based approach is presented for detecting change-prone classes, i.e., classes that change frequently (likely to be changed again in the future), and clusters of classes that frequently change together. The relationship between classes that change frequently together is termed change-coupling. Visualization is used in understanding these clusters of classes. The following research questions are investigated,

• Is it possible to identify and visualize the most change-prone collection of classes in an OO system?

• Is it possible to distinguish between local change-proneness (i.e., changes within a class) and change-proneness due to change-couplings (i.e., changes across classes)?

• Is the change-proneness due to change-couplings limited to the relations between classes in the logical design (including the use of design patterns) or does there exist other relations that are not explicitly represented in the design?

• How to visualize the change-prone information?

Class-level metrics such as number of attributes, total number of operations, depth of inheritance, and number of descendents are used to distinguish the characteristics of change-prone and non-change-prone classes, and identify design relationships (e.g., generalization.) The patterns are detected by the inspection of source code, reverse-engineered UML class diagrams, and documentation. Only intentional (i.e., well-documented) patterns are considered here. Metrics for change-proneness are defined to
detect local change-proneness and change-couplings of change-proneness. These metrics are computed from the logs of the version-control system. Box-plot outlier analysis is used to produce thresholds for the metric values indicating change-proneness.

A case study is described on an industrial application written in C++. Two versions, identified as A (first stable version) and B (latest version) are considered. Version A consists of 199 classes and 24 KLOC. Version B consists of 227 classes with about 32 KLOC. There are 37 intermediate versions and 191 “common” classes (possibly modified) between versions A and B. These “common” classes in version A were examined to predict the changes in version B. The local change-prone classes were found to be 36 out of 191 classes, the co-change coupling pairs were found to be 29 out of 924 co-change pairs, and the sum of pair couplings were found to include 29 out of 191 classes. Overall, 17 classes were found as change-prone classes that meet the metrics thresholds.

The five out of 17 (i.e., 29%) change-prone classes were involved in one or the other design pattern. The remaining 12 change-prone non-pattern classes form 7% of the non-pattern classes. The change-prone classes and the change-coupling between them are visualized with an architecture diagram which is very similar to a class diagram in terms of notations. The visualization revealed that there were change-coupling relationships between classes that were not represented by any design relationships. The class-level metrics reveal that change-prone classes are changed more frequently (on average 10 times more) and have more attributes and operations than non-change-prone classes. Not much difference was observed in the remaining class-level metrics.
Types of Changes and Origin Analysis

The tool *Beagle* (also discussed in Section 2.7.7) contains an analysis component for determining whether entities were added or deleted from one version to the next, and was used to perform origin analysis by Tu and Godfrey [Tu, Godfrey 2002]. Evolution metrics LOC, S-complexity, cyclomatic complexity, and number of function parameters, etc. are measured for each entity in a release (or version) and stored in a vector of evolution metrics. The similarity between two entities in different versions is represented by the Euclidian distance between their vectors. The similarity values are used for origin analysis of a given entity, i.e., lesser the distance, more the chances of an entity in a current version originating from the other entity in a previous version. An algorithm based on origin analysis and entity-name matching is used to identify the added and deleted entities between versions. The authors termed this form of origin analysis *Bertillonage analysis*. Another origin-analysis technique termed *dependency analysis* is also discussed. This technique is based on the hypothesis that the clones introduced by moves and renames may continue to honor many of the original relationships (e.g., calls, called-by, inherits, uses) exhibited in the previous version. A case study is demonstrated on two versions of the parser subsystem of *GCC*, *GCC 2.7.2.3* and *EGCS 1.0* (which is derived from *GCC 2.7.2.3*), to show the application of the origin analysis and differencing technique. The goal was to study the old architecture that continued to exist in *EGCS 1.0*. 
2.7.6 System Complexity

Capiluppi et al. [Capiluppi, Morisio, Ramil 2004] presents an approach studying the complexity of a software system. The complexity is measured in terms of changes in the system size (i.e., number of files and directories per release) and changes in the physical structure of files and directories (i.e., depth and width of the tree structure.) The objective of this work is to test hypotheses regarding the evolutionary characteristics (i.e., functional size grows over releases, the structure changes in uniform patterns, potential co-relationship between new developer arrival rate and code growth) of open-source systems. A case study is described on the ARLA system, an implementation of the AFS distributed file system. The latest release consists of 150 KLOCs and overall 45 developers participated in this project. The results show that the number of files and folders grow linearly with a superimposed ripple and their average sizes tend to stabilize over releases. The depth of the structure was approximately held constant while the width followed a trend similar to that of number of folders. This indicates that ARLA was a well-structured system right from its early inception in the software repository. It was also found that the on an average new contributors (i.e., contributions limited to a single file) have a higher arrival rate than the new authors (i.e., contribute multiple times and multiple files.)

Validation of Defect Detectors

The metrics and defect data available from the NASA’s Metrics Data Program (MDP) (collected for almost 8 years in some cases) are utilized by Menzies et al.
[Menzies et al. 2004] to address the following concerns that are typically raised by researchers regarding defect detectors based on the historical data,

- Lack of external validity – Do the defect detectors built from the data of one project scale to others?
- Buy, not build – If general conclusions about defect detectors (across projects) can be made or are available, why maintain project history anymore?
- Are static code measures such as Halstead/McCabe metrics “good enough” for such a task?

The authors describe their study on five NASA applications. Various data-mining tools LSR, M5, J48, and ROCKY are used to automatically generate defect detectors. The results obtained from these data miners are compared using the DELPHI approach (i.e., human-experts view.) Assessment measures such as precision, recall, accuracy, and effort in terms of LOC were used to study the variations (mean and standard deviation) in the output produced by the detectors for each project. It was found that these differences were very small. The results of a defect detector are improved at different stages in the project life cycle by using data from the local history. Furthermore, the authors suggest the use of static-code metrics as secondary indicators.

*Predicting Post-release Failures*

Nagappan et al. [Nagappan, Ball, Zeller 2006] used a combination of software complexity metrics and post-release defect history to build a predictor model for post-release failures in modules. The authors use five Microsoft products to validate the following four hypotheses that are paraphrased:
H1 - Higher complexity of a software entity statistically correlates to the number of defects reported post release.

H2 - A subset of metrics that satisfy H1 are applicable to all projects.

H3 - Post-release defects in new entities introduced in the same project can be predicted significantly by a metric combination.

H4 - A metric combination derived in H3 of a project can also predict entities that are likely to exhibit failures in different projects.

The hypotheses H1 and H3 found support in their study. A set of complexity metrics that correlates with post-release defects was found for each project (not the same set for all projects.) A regression model was build via Principal Component Analysis (PCA) to predict post-release defects. The hypothesis H2 was rejected. The hypothesis H4 was partially supported. H4 was only supported for projects that have the same or similar defect distribution. The authors caution the use of metrics in predicting post-release defects without assessing their applicability to the subject project. They recommend using the metrics that are validated with historical data.

2.7.7 Visualization

Information visualization is the use of computer-based, interactive visual representation of data to amplify cognition. A number of efforts in software visualization have been taken to use information-visualization techniques to support software maintenance and evolution. Software visualization approaches are typically very task specific [Maletic, Marcus, Collard 2002]. Here we examine a number of works specifically focused on the task to visualize the information mined from software
repositories. These approaches rely heavily on the visual presentation of the information in assessing the mined data and as such we group them together as a separate approach category.

Co-changing Files

Van Rysselberghe and Demeyer [Van Rysselberghe, Demeyer 2004b] propose a 2D visualization technique to recognize the change-relevant information from the log data in the CVS software repository. Files are mapped to the x-axis and time mapped to the y-axis. A change is represented by a “dot”, if a particular file has a change recorded (i.e., involved in a CVS delta) at a given time. Here, the change-relevant information of interest is the visual patterns identifying unstable components (under almost continuous change), coherent (co-changing) entities, design and architectural evolution (change in the relations between co-updating entities), and fluctuations in team productivity (heavy changes/ almost no changes in a given period of time.) A case study on the CVS version history of an open-source project, Tomcat, is also described. The found visual patterns indicative of the above aspects are validated with the available design documents and mail archives. The authors conclude that this visualization technique was helpful in understanding the evolution of the system, and locating further information about changes (e.g., developer communication regarding major design discussions.)

Another visualization of the clusters of frequently occurring co-changes is presented by Beyer and Noack in [Beyer, Noack 2005]. The co-changes derived from the log files of a version-control system are represented as an undirected bipartite graph. This graph is termed a co-change graph. An undirected edge is drawn from an artifact
node to a transaction node if an artifact is involved in that transaction or vice versa. An edge-repulsion LinLog energy model, which is an energy-based (or force-directed) graph layout producing method, is used to layout the co-change graph. A formalism is presented on the idea of having a small distance (i.e., short edges) between artifacts that participate together in a large number of transactions and a large distance (i.e., long edges) between artifacts that participate together in a relatively few number of transactions. The artifacts that are placed together in the layout give an impression of a cluster. The approach is evaluated on the three systems CrocoPat 2.1, Rabbit 2.1, and Blast 1.1 consisting of variety of documents (e.g., source code, build files.) The details about the size and the historical data used are presented in a table in [Beyer, Noack 2005]. These statistics were collected with the help of a tool StatCvs. The tool cvs2cl extracted transactions from the CVS logs, and the tool CrocoPat generated the co-change graphs at a file level. The layouts were automatically computed using the Barnes-Hut algorithm and the edge-repulsion LinLog model. The clusters obtained in this layout were compared to the authoritative decomposition (e.g., subsystems) of the system. Nodes (i.e., files) belonging to the same subsystems are given the same color. In conclusion, most of the clusters in the layout confirmed with the authoritative decomposition such as subsystems. However, there were instances where artifacts cannot be assigned to a unique subsystem, and there was no clear separation of different subsystems.
Structural and Architectural Changes

The tool *Beagle* provides two simultaneous views referred to as structure diagrams and dependency diagrams [Tu, Godfrey 2002]. The structure diagram is a hierarchical (tree) view with software entities such as subsystems (i.e., directories) and modules (i.e., files) mapped to the internal nodes, and functions mapped to the leaves. Colors and saturations are used to encode difference information (e.g., additions and deletions) and age of entities (e.g., a lighter shade indicates more recent changes.) This view helps in understanding the structural changes that occurred between two arbitrary releases.

The dependency diagram shows the architectural difference between two releases. The architecture is defined as the above mentioned software entities and the relationships (e.g., new call, new reference, delete implemented by) between them. This diagram helps to visualize the architectural difference at various (physical) levels of granularity (e.g., subsystems and files.) The structure diagram forms the navigation component for selecting an entity (e.g., subsystem) of interest and the dependency diagram forms the detail component for examining the architectural differences within the selected entity (i.e., the contained files and the relationships between them) across given releases. A case study describing the versions comparison and evolution visualization between *GCC V2.0* and *GCC V2.7.2* is described in [Tu, Godfrey 2002]. For an instance of a view showing a structural diagram and dependency diagram, we refer the readers to Figure 5 in [Tu, Godfrey 2002].
Holt and Pak [Holt, Pak 1996] presents a visualization tool, namely $GASE$, for representing software structural changes. The software system is represented as a 2D graph. The nodes represent the modules and the edges represent the relationships such as calls or includes. Further drill-down of the nodes reveals their sub-modules and the relationships between sub-modules. The tool incorporates fact extraction to construct a 2D graph and difference analysis to identify changes. Colors are used to show the differences in the nodes and edges between two versions of a software system. The tool is applied on eleven versions of an industrial system with over four years of development history. The subject system was written in C and each version consisted of about 80 KLOC. The observations indicate that the tool was effective in identifying restructurings, consistent growth of the subject system, undocumented and unknown structural dependencies, and the existence of a “software rule” which states that the rate of change is directly proportional to the structural depth, i.e. most changes occur within modules and not at a subsystem level. All these observations were verified with the developers of the subject system.

Gall et al. [Gall, Jazayeri, Claudio 1999] presents an interesting 3D visualization technique in order to simultaneously view an attribute of the structure of a software system across multiple releases. The change information of a program (and other properties such as size and complexity metrics) is represented by an attribute that contains the value of the release number in which it last changed (i.e., added, modified, or deleted.) Different 3D shapes (e.g., spheres and cubes) are used to distinguish nodes representing system, subsystems, modules, and programs. Each release of a system is
represented by a 2D tree ordered by release numbers. This forms a 3D diagram with a tree (i.e., x and y axes) for each release number (i.e., z axis). Color spectrum (i.e., a customized rainbow scale) is used to choose (successive) colors equal the number of releases of a system. The nodes corresponding to programs, modules, and subsystems, are displayed in the appropriate colors based on their release attributes. The technique is demonstrated on 20 releases of an industrial system over a period of two years. The subsystem system consisted of 8 subsystems, 47 to 50 modules, and 1,500 to 2,300 programs. For further information on the various views and observations reported on the subject system, we refer the readers to [Gall, Jazayeri, Claudio 1999].

Change Smells and Refactorings

A graph visualization with nodes representing classes and edges representing logical couplings is used to identify change smells by Ratzinger et al. [Ratzinger, Fischer, Gall 2005]. The notion of change smells based on the strength of logical couplings between entities is presented with an analogy to bad smells as introduced by Fowler [Fowler 1999]. Change smells are considered as indicators of structural deficiencies that are candidates for reengineering based on the change history. Refactorings based on two change smells, namely man-in-the-middle and data containers are discussed. Standard refactorings such as Move Method and Move Field are suggested to alleviate the man-in-the-middle problem. Refactorings such as Move Method and Extract Method are suggested to improve the code exhibiting data-container smell. A case study is described on an industrial Picture Archiving and Communication System (PACS) with 500,000 lines of Java code. The change history of 15 months in the CVS repository was used to
identify the man-in-the-middle smells with the help of the visualization. The suggested refactorings were applied and the logical couplings were observed again after a period of another 15 months. The authors discuss one such case of an ImageFetcher class showing smells of man-in-the-middle. It was observed that the logical coupling between ImageFetcher and other associated classes decreased substantially at the end of the 15 month observation period.

The (end result of) refactorings that are detected with the technique described in [Görg, Weißgerber 2005b] are visualized in Görg and Weißgerber [Görg, Weißgerber 2005a]. The detection and visualization of structural refactorings (Move Class, Move Method, Pull Up Method, Push Down Method), and local refactorings (Hide Method, Rename Method, Add/Remove Parameter) is demonstrated on the jEdit and Tomcat projects. The visualization provides class-hierarchy and package-layout views. Different colors are used for representing different kinds of refactorings. UML symbols are used for representing both class and the relationships (e.g., generalization.) The relationship symbols are appropriately colored to indicate their part in the corresponding structural refactorings.

Visualizing Data Mining Rules

Burch et al. [Burch, Diehl, Weißgerber 2005] discuss techniques to interactively visualize association and sequence rules mined from software archives by using data-mining techniques. They further present views that combine the static structure of items (i.e., files) with the temporal order in a rule. Views such as pixel-map, parallel coordinated view, rule matrix, and support graph are realized in the tool EPOSee.
Examples of these views are demonstrated on the Mozilla repository. Clusters and outliers of changed files identified in these examples are discussed.

2.7.8 Clone-Detection Methods

Simply stated, source code entities with similar textual, structural, and/or semantic composition are referred to as source code clones. A number of approaches exist in the software engineering literature that addresses identification of both exact and near-miss clones. Simple approaches such as text-based and token-based techniques have been applied with a reasonable degree of success. Other approaches operate on source code abstractions such as Abstract Syntax Trees (AST) and Program Dependency Graphs (PDG.) In this subsection, we discuss application of clone detection techniques in the context of MSR. Our discussion is organized with regards to the purpose of MSR (i.e., level-two subsections.)

Clones and their Relationships

An approach based on the history of source code clones is presented by Kim and Notkin [Kim, Notkin 2005] to assist in maintenance. A clone-detection tool, namely CCFinder (www.ccfinder.net), is used to identify clone groups in each version of a program in the CVS repository. A cloning relationship is assigned to the corresponding clone groups in the consecutive versions. The cloning relation is assigned a singleton value based on the type of a change performed (e.g., add is assigned if at least one element is inserted in a clone group.)
The nodes (clone groups) and edges (cloning relationships) form a directed graph, termed clone lineage. The code lineage is obtained on identification of all clone groups and the cloning relationships between each consecutive versions. A set of clone lineages originating from a same clone group is termed clone genealogy. Using the Clone genealogy information, the following questions are investigated,

- How many source code clones impose a serious maintenance challenge?
- Is aggressive refactoring always the best solution for maintaining (i.e., eliminating) clones?

The investigation of these questions is carried out on two Java open-source projects, *carol* (library to use different implementations of RMI) with 23,731 LOC as of October 2004 and *dns-java* (DNS server) with 20,752 LOC as of June 2004. The file versions that contain clones were analyzed i.e., 37 (out of 164) versions for *carol*, and 27 (out of 39) versions for *dns-java*.

The number of clone genealogies found in *carol* and *dns-java* were 109 and 76 respectively. The clone genealogies were further analyzed to determine the number of consistent clone genealogies (i.e., all the lineages in the genealogy are consistent.) The number of consistent clone genealogies found in *carol* and *dns-java* were 41 (38%) and 24 (32%) respectively. These results indicate that clones were required to be, or were maintained (and not eliminated) during the evolution of the considered systems.

The clone lineages and clone genealogies were marked as “locally factorable” or “locally unfactorable”. For further information on “locally factorable” and “locally unfactorable”, we refer the readers to [Kim et al. 2004]. The investigation of the above...
questions is carried out on two Java open-source projects, *carol* (library to use different implementations of RMI) and *dns-java* (DNS server.) The number of “locally unfactorable” clone genealogies found in *carol* and *dns-java* were 70 (64%) and 52 (68%) respectively. The examination of the clone genealogies that existed for more than 20 versions, 37 in *carol* and 11 in *dns-java* revealed 19 in *carol* and 3 in *dns-java* were both consistently maintained and “locally unfactorable”.

Further investigation on why the clones were maintained, it was found that out of the 53 dead genealogies (eliminated in the most current version) in *carol*, 42 were eliminated in less than 10 versions. In the case of *dns-java* it was found to be 41 of the 59 genealogies. The authors’ hypothesis that such a behavior exists due to the programmers’ preference for not committing to a particular design abstraction when dealing with the volatile design decisions. The manual inspection of the two system found that about 25% to 48% of the clone lineages were actually diverged (possibly due to refactorings) from its original place (group) to some other location (group.) Therefore, they are not completely eliminated.

*Frequently Occuring Changes*

The concept of Frequently Applied Changes (FACs) is introduced by Van Rysselberghe and Demeyer [Van Rysselberghe, Demeyer 2004a]. The FACs are defined as changes occurring multiple times in the version history of a system. All the *CVS* deltas are examined (via *cvs log* command) and their corresponding source code changes (via *cvs diff* command) are recorded in a text file. A clone-detection tool, *CCFinder*, using parameterized token matching is applied to this text file to find similar pairs of
source code changes (i.e., clones.) The CVS deltas corresponding to these clones are considered as the FACs. This technique is evaluated on the three year version history of an open-source system, *Tomcat*. Both high and low threshold values of the number of matching tokens are experimented with to detect FACs. High threshold values produced a small set of clones that were almost identical. It was observed these FACs were typically caused by a “well-established” solution at one place being replicated at other locations (later eliminated by a function), moving code (considered deleted and then added), and temporary addition of code that was later deleted. The authors suggest that such FACs are indicators of the reasons for code duplication, possible design improvements, and the situations in which temporary solutions can be adopted. A more rigorous similarity comparison was employed to declare clones as FACs. The changes were considered as FACs, if both the code before and after the change were clones. The authors suggest such FACs may be used to identify recurring change patterns and in turn identification of refactorings.

*Code Duplication and Origin Analysis*

In another approach by Godfrey et al. [Godfrey et al. 2004], both parameterized and metrics-based string-comparison techniques are used to study code duplication within the file-system component of the *Linux* operating system. Additionally the clone-detection methods and the fact-extraction tool *cppx* were used to perform origin analysis of the parser subsystem of *GCC* (*EGCS* variant, version 1.0.) Entities are often moved and renamed (with possibly some other changes) during the evolution of a system as a result of code restructuring or redesign (e.g., refactoring.) If such moves and renames are
not identified, they are reported as deletions and additions of the “same” entity (possibly multiple times.) Therefore, the “true” origin of an entity is lost. The clone-detection technique is used to identify such moves and renames. However, the downside is that the reported candidates may be “real” artifacts of cloning. The authors hypothesis is that the clones introduced by moves and renames may continue to honor many of the original relationships (e.g., calls, called-by, inherits. uses) exhibited in the previous version. The cppx fact-extraction tool is used to facilitate such relationship analysis. No information is available on how the approach is evaluated.

2.7.9 Frequent Pattern Mining

The field of data mining provides a variety of techniques for discovering implicit knowledge from a large dataset such as patterns, trends, and rules. In a very broad sense, data mining encompasses information retrieval, statistical analysis and modeling, and machine learning. However, each is a separate field having applications to MSR. Therefore, instead of covering all these fields under a common umbrella of data mining, we discuss each on its own. Frequent-pattern mining is one such data-mining approach that has been used in MSR. Itemset mining and sequential-pattern mining have been applied to uncover software entities that frequently co-change i.e., frequent patterns. Itemset mining precludes ordering information, whereas, sequential-pattern mining includes ordering information, of changed entities forming a pattern. We now discuss such mining techniques that utilize the metadata, source code data, and difference data found in the software repositories.
Evolutionary Couplings and Change Predictions

Zimmermann et al. [Zimmermann et al. 2004] aim to identify co-occurring changes in a software system. The purpose is to find when a particular source code entity (e.g., function with name $A$) is modified what other entities are also modified (e.g., functions with names $B$ and $C$.) The presented tool, $ROSE$, parses the source code (C++, Java, Python) to map the line numbers to the syntactic or physical-level entities [Zimmermann et al. 2004]. The subsequent entity changes in the $CVS$ repository are grouped as a transaction using a sliding-window technique [Zimmermann et al. 2004]. An association-rule mining technique is employed to determine rules of the form $B \Rightarrow A$. Examples of deriving association rules such as a particular “type” definition change leads to changes in instances of variables of that “type” and coupling between interface and implementation is demonstrated. Their technique has various capabilities:

- Ability to identify addition, modification, and deletion of syntactic entities without utilizing any other external information (e.g., AST.)
- Handles various programming languages and HTML documents.
- Detection of hidden dependencies that cannot be identified by source code analysis.

An extension to this work is reported in [Zimmermann et al. 2005] that allows prediction of additions to and deletion from entities. The tool $ROSE$ is evaluated for navigation (recommendation of other affected entities), prevention (find missing changed entities after a developer declares a transaction complete), closure (false suggestions for missing entities), granularity (fine vs. coarse), maintenance (modified only),
multidimensional (addition and deletion), history, and recent changes. Eight open-source projects are considered with an evaluation period of at least a month selected for each project. For a given project, the changes that occurred during the evaluation period were predicted based on previous versions. Additional measure feedback (percentage of queries that resulted in at least one recommendation) is introduced to assess the “interactive power” of the ROSE tool.

The average precision, recall, and feedback values taken across the given eight projects for

- navigation support with fine granularity are 29%, 33%, and 66% respectively, whereas, navigation support with coarse (file-level) granularity are 29%, 44%, and 82% respectively,
- prevention support with fine granularity are 69%, 75%, and 3% respectively, whereas, prevention support with coarse (file-level) granularity are 70%, 76%, and 7% respectively,
- navigation support with fine granularity and maintenance transactions are 30%, 44%, and 71% respectively, whereas, navigation support with fine granularity and non-maintenance transactions (at least one item added/deleted) are 29%, 25%, and 63% respectively.

The average feedback values in case of closure are 1.9% and 3% for fine and coarse granularity respectively. The tool ROSE needs only a few weeks of history to make suggestions in the close vicinity of the above reported assessment values.
Furthermore, the results can be improved by assigning higher weight to recent changes in case of projects undergoing rapid renames and moves.

A similar approach is taken by Ying et al. [Ying et al. 2004] for source code change prediction at a file level. An association-mining technique based on FP-tree itemset mining is used. The mined rules are classified into categories surprising, neutral, or obvious to indicate their interestingness. The technique is evaluated on the version histories of Mozilla and Eclipse projects.

**Mining Usage Patterns**

Livshits and Zimmermann [Livshits, Zimmermann 2005] present an approach based on itemset mining for discovering call-usage patterns (e.g., call pairs and state machines for more than two method calls in an object) from source code versions. In addition to the standard ranking methods typically used in data mining, they present a corrective ranking (i.e., based on past changes that fixed bugs) to order the mined patterns. The objective of this work is to determine useful usage patterns and their violations. The hypothesis is that violations of useful patterns are potential sources of errors. The patterns are classified into valid patterns, likely error patterns, and unlikely patterns. A snapshot of the source code is instrumented to obtain the run-time information of method calls. A candidate pattern mined from the version archive is considered to be a valid pattern if it is executed a specified number of times and an unlikely pattern otherwise. Likewise, if a valid pattern is also violated (i.e., only a proper subset of the calls are executed) a (larger) number of times, it is considered as an error pattern. The approach is validated on Eclipse and jEdit systems. The results indicate that
their approach, along with the corrective ranking, was effective in reporting error patterns.

While the above work used itemset mining (or association mining), sequential-pattern mining have also been used to the problem of uncovering frequently patterns of co-changes.

**Ordered Change Patterns**

Kagdi et al. [Kagdi, Yusuf, Maletic 2006] applied sequential-pattern mining to uncover frequently changed files with the supplementary information of their change order. Modern source-control systems, such as Subversion, preserve change-sets of files as atomic commits. However, the ordering information in which files were changed in a change-set is typically not recorded in source code repositories. They define six heuristics for grouping the “related” change-sets in a source code repository. Given such groups, sequences of files that frequently change together are uncovered using sequential-pattern mining. For example, sequences of changed-files such as \{f1\}→\{f2\} and \{f4\}→\{f5\} are uncovered. The sequence \{f1\}→\{f2\} indicates that the changes in \{f1\} *happens before* the changes in \{f2\}. This approach not only gives the (unordered) sets of files but also supplements them with (partial) ordering information. Therefore, this approach of changed-files sequence-mining subsumes the approach of changed-files itemset mining. Their technique is demonstrated on a subset of KDE source code repository. In other works, Burch et al. [Burch, Diehl, Weiβgerber 2005] presents a tool that supports visualization of association rules and sequence rules, El-Ramly and Stroulia [El-Ramly, Stroulia 2004] used sequence mining to detect patterns of user activities from
the system-user interaction data, and Xie and Pei [Xie, Pei 2006] used sequence mining to filter the results of a source code search tool to report API-usage patterns in which a source code entity is used.

2.7.10 Information Retrieval (IR) Methods

Information Retrieval (IR) is another methodology that is used for classification and clustering of textual units based on various similarity concepts. IR methods have been applied to many software engineering problems such as traceability, program comprehension, and software reuse. Metadata such as CVS comments, textual descriptions of bug reports, and emails makes IR an attractive choice. In this subsection, we discuss IR techniques applied to MSR.

Classification Based on the Cause of a Change

An IR-based method for the classification of MRs with regards to the purpose of a change is presented by Mockus and Votta [Mockus, Votta 2000]. An automatic keyword clustering and classification (heuristic-based) algorithm is applied on the textual description of a MR and the text messages of the associated deltas (i.e., commit operations) in the version-control system. Here, the considered change-management system, Extended Change Management System (ECMS), records explicitly the MR associated with each delta. The authors preliminarily focused on the three types (purposes or reasons) of a change: adding new features (adaptive), fixing bugs (corrective), and code restructuring for future changes (perfective.) An additional category, inspection was discovered from the initial results of the algorithm on a test
system. Further interest was on studying the relation between the type, size, and time-effort of a change.

A proprietary telecommunication subsystem was used as a test case to demonstrate the classification approach and investigate the following questions:

- How does the purpose of a change relate to size and interval (time-effort)?
- How does the purpose of a change related to perceived difficulty by the developers?

The method was able to automatically classify 88% of the MRs with corrective, perfective, adaptive and inspection forming 33.8%, 3.7%, 45%, and 5.3% respectively. The unclassified 12% MRs were later inferred to be corrective type from the manual validation by the authors. The percentage of the total number of deltas, lines added, lines deleted, and lines left unmodified are

- Corrective: deltas 22.6%, added 18%, deleted 18%, and unmodified 27.2%,
- Perfective: deltas 4.3%, added 3.5%, deleted 5.8%, and unmodified 4.5%,
- Adaptive: deltas 55.2%, added 63.2%, deleted 55.7%, and unmodified 48.3%, and
- Inspection: deltas 8.5%, added 5.4%, deleted 10.8%, and unmodified 10.3%.

The above numbers give an idea between the type and the size of a change. All the changes are not identical and vary in the size (expressed in the above attributes) with regards to the type. From the time-effort point of view, corrective changes were found to be of shortest interval, followed by perfective changes. The 35% of the most time consuming adaptive changes took considerably longer than their corresponding inspection changes. On the lower end, the 60% of least time consuming inspection
changes took considerably longer than their counterpart adaptive changes. The authors attributed this disparity due to the need of formal inspection for changes extending more than 50 LOC.

The level of difficulty (easy, medium, hard) perceived by developers was collected for 170 changes. The results indicate corrective changes are perceived to be mostly likely hard, followed by perfective changes however, the inspection changes are perceived easy.

The quality of the automatic classification was validated with the developers’ opinion. The results of the automatic classification of a selected few (30-150) MRs were validated with the manual classification performed by the developers. About 61% of the time they were in agreement with each other.

Change Prediction

Canfora and Cerulo [Canfora, Cerulo 2005] used the bug descriptions and the CVS commit messages for the purpose of change predictions. Their approach provides a set of files that are likely to change based on only the textual description of a newly-introduced bug (or feature) in the bug repository. An information-retrieval method is used to index the changed files in the CVS repositories with the textual description of past bug reports in the Bugzilla repository and the CVS commit messages. A bug report is linked to a CVS commit (i.e., a set of changed files) based on the explicit bug identifier found (a common practice in open-source development) in that commit message (e.g., bug id 30,000.) The corpus resulting from this method is used to query for a list of
relevant files that are likely to change due to a given bug report. The query is formed from the textual description of a bug report.

The approach is evaluated on four open-source projects *Kcalc*, *Kpdf*, *Kspread*, and *Firefox*. Precision and recall metrics are used as the assessment metrics. A validation technique known as leave-out-one is used. That is, the indexing is formed on all bug reports (typically already fixed) except the one whose change-set is estimated. The estimated files produced by the method are used to compute precision and recall metrics. Precision and recall were found to increase with larger amounts of textual information, e.g., complete versus short descriptions of bug reports. For the projects *Kcalc* and *Kspread* the bug descriptions performed better than CVS commits, whereas, the inverse behavior was found in *Kpdf* and *Firefox*. Precision and recall are reported for 30 queries on each project. The precisions of *Kcalc*, *Kpdf*, *Kspread*, and *Firefox* are reported in the range [38%, 78%], in the range [36%, 45%], 39% and 36% respectively. The recall of *Kcalc*, *Kpdf*, *Kspread*, and *Firefox* are reported in the range [82%, 98%], in the range [70%, 85%], 79% and 67% respectively.

A further extension of this work is reported in [Canfora, Cerulo 2006] where the prediction is indexed at a line-level granularity of source code. The evaluation on three open-source projects *Gedit*, *ArgoUML*, and *Firefox* shows over 10% improvement in precision compared to file-level granularity. However, the cost of indexing at line-level is in the order of hours compared to the order of seconds with file-level granularity.
Importance of Human Guidance

The importance of a human analyst in refining the data produced by data-mining tools, and further guiding and tuning the data-mining process is argued by Hayes et al. [Hayes, Dekhtyar, Sundaram 2005]. It is typical of data-mining tools not to produce “perfect” results (i.e., both precision and recall are never 100%). Such results may create (negative) ripple effects when utilized to help automate a desired task. In order to deal with this problem, the authors suggest that only the refined results obtained by an analyst (and not the ones directly produced by a data-mining tool) should be made available to others (i.e., tools/human.)

Two case studies [Hayes, Dekhtyar, Sundaram 2005] on the MODIS dataset are described and reported the following questions:

- Are the better (refined) accuracies of both the analyst and tool equivalent?
- Are there any other factors that affect analyst decision-making? Level of expertise? Trust of the software?

A pilot study is conducted on the MODIS dataset consisting of 19 high-level requirements, 49 low-level requirements, and 41 true links between them. A traceability tool, SuperTracePlus, based on an information retrieval (IR) technique is used. The data obtained by this tool were refined by experienced analysts (i.e., highly familiar with the tool but only slightly familiar with the domain.) The results indicate that further

---

3 Here, the term “data mining” is used in a broader context, including information retrieval (IR) methods.
refinement by the analysts increased precision but decreased recall. However, no general conclusions were deduced considering the small size and scope of the case study.

In another study, a traceability task was assigned to experienced analysts using the SuperTracePlus tool with three different settings of precision and recall values (one setting per analyst.) They were asked to report both the original and refined precision and recall values along with the time spent in a one-week period. The results of this study show that data sets with low recall took relatively longer time to complete and produce no worse final (refined) results than data sets with high recall. Overall, no performance-improving pattern was discovered. While this work is not directly applied to multiple versions of a software system, we feel that it contributes a useful, and efficient, technique for analysis of multi-version repositories.

New Developer Assistance

Cubranic et al. [Cubranic, Murphy 2003; Cubranic et al. 2005] describe a tool, Hipikat, to assist new developers (not necessarily novice) on a project. Various artifacts (e.g., source code, email, and bug reports) produced in the project are integrated to form a project memory. A vector-based IR method is used to draw the similarity between artifacts. Other relationships between artifacts are formed by using heuristics, e.g., MRs in Bugzilla are related to the files in CVS by matching bug-id in the commit messages. Hipikat recommends artifacts from the project memory that may hold relevance to a task at hand. A developer may ask for the relevant artifacts explicitly in the form of a explicit query, or the tool can do so automatically based on the current context (e.g., based on the currently open documents in the developer’s workspace.)
Two studies are discussed in [Cubranic et al. 2005] to validate the tool. One study focuses on the quality of Hipikat’s recommendations. Twenty bugs from the Bugzilla database of Eclipse were randomly selected from 215 bugs that were assigned a severity status of “minor” and fixed between June 2002 and March 2003. The task was to recommend the relevant source code files to these bugs. The maximum precision and recall values were found to be 56% and 71% respectively. The average precision and recall values were found to be 11% and 65% respectively. The minimum values for both precision and recall were reported to be 0 in the case of four bugs.

The other is a usability study for new developers (with software tool and development experience) using the Hipikat tool. For details on user tasks and questions we refer the readers to [Cubranic et al. 2005]. The results show that Hipikat was used more (i.e., accessed and queried more) in the initial understanding of the assigned task but not in the execution of it. The participants utilized the recommendations (and possibly more queries based on them) until they got to a starting point providing access to the relevant source code. Venolia [Venolia 2006] proposed a similar tool that allows a full-text search on different artifacts stored in various software repositories. This includes processing of information such as email addresses and URLs found in unstructured text.

Commonly Occurring Phenomena

Time-series representation and frequency-domain analysis approaches have been proven successful in domains such as image/speech processing and stock-market forecasting to detect commonly occurring similar phenomena that evolves over time.
The applicability of one such efficient approach, Linear Predictive Coding and Cepstrum coefficients (LPC/Cepstrum) for compact representation of the evolution of software modules is examined by Antoniol et al. [Antoniol, Rollo, Venturi 2005]. The size of a software module changing (typically increasing) over versions is thought of as a time series and is represented by an ordered set of LPC coefficients. The LPC coefficients are used to compute Cepstrum coefficients by inverse Fourier transformation. The LPC/Cepstrum series provides the approximation of the time-evolving series and preserves most of the relevant information of the original series. All the evolving software modules are transformed to their respective LPC/Cepstrum representations. A comparison measure such as Euclidian distance is used to compute the “closeness” (similarity) between LPC/Cepstrum representations of software modules. A distance threshold is defined, below which the measures are considered similar.

The LPC/Cepstrum approach is applied to the Linux kernel versions 1.1.0 to 1.3.1.0 consisting of 1,788 files and 211 releases. The size of the modules was defined by LOC metric. The LPC/Cepstrum computation was performed in less than 5 seconds on a P4 1.6 GHz machine. Increasing the LPC/Cepstrum series length (better approximation of the time series) resulted in decreasing number of similar pairs. Similarly, increasing the threshold requirement (e.g., $10^{-3}$ to $10^{-5}$) resulted in decreasing number of similar pairs. Different combinations of the LPC/Cepstrum series lengths and distance threshold values were also tested for similarity detection. It was found that more similar pairs were reported with decreasing both the LPC/Cepstrum series length and the threshold.
2.7.11 Classification with Supervised Learning

The term Machine Learning refers to techniques that are capable of automatically acquiring and integrating knowledge in order to improve performance for the desired task(s.) Supervised learning is a technique for creating a cause-effect function from training data. The training data is divided into the input objects and the desired outputs or classifications. The software repositories containing the historical data (and metadata) of an evolving software system allows machine-learning techniques to be applied for discovering and forming classification and prediction models within the context of MSR.

Maintenance Relevance Relations

A classification-learning technique is used by Shirabad et al. [Shirabad, Lethbridge, Matwin 2001; Shirabad, Lethbridge, Matwin 2003; Shirabad, Lethbridge, Matwin 2004] to determine the co-update relations between a pair of source code files i.e., given two files determine whether a change in one leads to a change in the other. Such types of relations are also termed maintenance-relevance relations. A decision-tree classifier (i.e., model) is produced by a machine-learning (induction) algorithm. A time-based heuristic is employed to assign a relevant or not-relevant relation between a pair of files to form the learning and testing sets. A fixed time period between time T1 and T2 (T2 < T1) is chosen and if a given pair of files changed together in any update during that time, the relation is considered relevant. Another time period between T3 and T2 is chosen (T3 < T1) and all the relations between a pair of files that are not marked as relevant are considered not-relevant. The classifier takes as input a pair of files and assigns the co-update relation between them to either relevant or not-relevant categories.
The files are described by their attributes divided into syntactic (e.g., function calls, variable, type definitions) and text-based types (e.g., text descriptions of PR, program traces, memory dumps, file comments.) Note that the text-based attributes are represented by a Boolean bag of words (all the possible values in a set of documents after the stop-word, transformation-list and collocation-list processing.) Comparing files based on text-based attributes is then reduced to performing a logical AND operation on a pair of Boolean vectors. The syntactic attributes are given by a list of name-value pairs.

The approach was validated on a telephone switching system with 4,700 files and 1.9 MLOC written in a high-level programming language and assembly language [Shirabad, Lethbridge, Matwin 2004]. Three classifiers were obtained based on the problem report (text), comment, and syntactic attributes. The analysis of the ROC (false-positive rate versus true-positive rate), precision, and recall plots imply that the problem-report attributes generate better classifiers than those of syntactic ones. The comment attributes generated classifiers do not perform at par with those generated with the problem report attributes. However, they are better than those generated from the syntactic attributes. The classifiers generated from a combination of syntactic and comment attributes produce better results than either of them considered alone.

**Triage Bug Reports**

Anvik et al. [Anvik, Hiew, Murphy 2006] used a supervised learning (i.e., Support Vector Machine algorithm) in order to recommend a list of potential developers for resolving a bug report. Past reports in the Bugzilla repository are used to produce a classifier. The authors develop project-specific heuristics to train the classifier instead of
directly using the *assigned-to* field of a bug report. This was done to avoid incorrect assignment of bug reports with default assignments that may not necessarily reflect the actual developer who resolved a bug. The approach is evaluated on three open-source projects *Eclipse*, *Firefox*, and *GCC*. Developers that contributed at least nine bug-report resolutions over the most recent three months were considered in the training set for *Eclipse* and *Firefox*. The precision for *Eclipse* and *Firefox* was 57% and 64% respectively and the recall 7% and 2% respectively. The precision of *GCC* was 6% for recommending one developer and 18% for two/three developers. The recall of *GCC* was 0.3%, 2%, and 3% for recommending one, two, and three developers respectively.

### 2.7.12 Social Network Analysis

Social network analysis [Wasserman, Faust 1994] is a technique widely used in social and behavioral sciences for deriving and measuring of “invisible” relationships between social entities (i.e., people.) In the context of MSR, social network analysis is applied to discover developer roles, contributions, and associations in the software development.

#### Developers Roles and Contributions

An approach based on social network analysis to group developers using the logs (deltas) stored in the *CVS* repository is proposed by Huang and Liu [Huang, Liu 2005]. The log data is analyzed to determine developers’ contributions at a module (directory) level. This information is used to construct a graph where a node represents a developer and an edge represents a “common contribution” relationship. An edge exists between a
pair of developers if they are found to contribute deltas to the same directory. This graph is analyzed to find core and peripheral developers based on the distribution of the distance-centrality values. The distance-centrality value of a node is basically the inverse of the summation of the distances between it and every other node. The lesser the distance-centrality value of node, the more the connection with (possibly many) other nodes. The authors report their findings on six projects selected from SourceForge. In one project, all the developers were found to have similar roles. They found core developers (indicated by high distance-centrality values) formed a relatively small group, controlled the source code, and played central roles. The other peripheral developers made minor contributions. The core members were shown to work very closely with each other. The peripheral developers were found to rarely work with the other peripheral contributors.

A similar approach was earlier described by Lopez-Fernandez et al. [Lopez-Fernandez, Robles, González-Barahona 2004] to construct committer networks (i.e., vertices are mapped to committers and edges are mapped to contributions to a common module) and module network (i.e., vertices are mapped to modules and edges are mapped to contributions by a common developer) from the CVS log data. Various graph characteristics such as degree of a vertex and clustering coefficient of a vertex are suggested and interpreted. A case study and the results are discussed on Apache, GNOME, and KDE systems.
Inter-Projects Collaboration

A visualization tool, *Graphmania*, with the goal of supporting cross-project knowledge sharing and collaboration is presented by Ohira et al. [Ohira et al. 2005]. The authors observe from the analysis of over 90,000 projects hosted on *SourceForge* that small projects typically consist of few developers (e.g., 66.7% of the projects had only a single developer.) The *Graphmania* tool is targeted to support developers involved in a small project for performing tasks by utilizing both the knowledge of developers of other projects and the relevant information from other projects. The authors believe that such a tool may encourage other non-contributors to turn into active participants by directly supporting the questions “Who should I ask?” and “What can I ask?”.

The *Graphmania* tool provides three types of collaborative social networks: developer networks, project networks, and developer-project networks. These networks are represented by an undirected weighted graph with the following mappings,

- **developer networks** - A node represents a developer and an edge between a pair of nodes (developers) represents their participation in at least one common project. The number of common projects is used to assign a weight to the edge.
- **project networks** - A node represents a project and an edge between a pair of nodes (projects) represents at least one common developer. The number of common developers is used to assign a weight to the edge.
- **developer-project networks** – All the nodes and edges of the developer networks and project networks. Additionally, edges are introduced between developer nodes and their corresponding participating project nodes (i.e., if a developer is a
participant of a project)

A case study describing the application of the Graphmania tool on the above dataset from SourceForge but limited to nodes with a maximum of five edges is described. Only the small subsets (sub-graphs) of the three networks are presented. Further analysis shows that the bridge nodes in the case of a developer network may reveal a “linchpin” developer connecting to the other component (social network) of a graph. Such “linchpin” developers may form potential contacts of external knowledge. In the case of a project network, a developer involved in a cluster of projects can share information and help avoid consideration of other irrelevant projects from other clusters. Similarly, the developer to project edges in the developer-project network may help a developer to acquire information for a given project (task) from other neighbors (i.e., other projects in which the same developer is involved.)

2.8 Discussion & Open Issues in MSR Research

The realistic nature (i.e., actual evolution data) of MSR investigations appears to be a promising avenue to help support and understand software evolution. However, establishment of history-based techniques as an alternative and/or complement to traditional techniques remains largely an open question for further investigation. Answering this question will provide the underlying validation of MSR research.

In order to take steps towards this the following issues needs to be addressed: 1) we need to be able to perform MSR on fine-grained entities; 2) there needs to be clear guidelines for the number of versions to be considered; and 3) standards for validation must be developed. Let us discuss each of these issues in more detail.
2.8.1 MSR on Fine Grained Entities

One major issue is the disparity between the software-evolution data available in the repositories and the needs of the stockholders, not just researchers but also including software maintainers. The majority of current MSR approaches operate at either the physical level (e.g., system, subsystems, directories, files, lines) or at a fairly high-level of logical/syntactic entities (e.g., classes.) This is regardless of the primary focus, i.e., changes of properties or artifacts. In part this is due to the researchers restricting their approaches/studies to what is directly available and supported by the software repositories (e.g., file and line view of source code and their differences.) However, the investigations by Zimmermann et al. [Zimmermann et al. 2005] have shown the benefits of further processing the information directly available from source code repositories for change prediction and impact-analysis tasks.

In their study [Zimmermann et al. 2005], there was no significant difference in precision and recall values between file-based and logical-based entities (i.e., classes, methods, and variables) with respect to change-prediction tasks. However, there is an implicit gain in terms of the context available to the maintainer, for example the exact location of a predicted change. Predicting a change at an entity-level rather than a file-level reduces the manual effort as only the predicted entities (versus the whole file) needs to be examined. This leads to the issue of extending current MSR by increasing the *source code awareness*.

The issue of source code awareness could be twofold with regards to the types of MSR questions and the source code artifacts and differences. For example, on one end, a
market-basket question (MBQ) is used to find logical/evolutionary couplings between
source code entities. These couplings are termed “hidden” dependencies as they are
solely based on historical information of software changes. However, very little attention
has been paid as to whether these hidden dependencies correspond to relationships
present in well-established source code models (e.g., control-flow graphs, dependency
graphs, call graphs, and UML models.) We feel that a finer-grained understanding of the
source code changes is needed to address these types of questions. Fluri et al. [Fluri,
Gall, Pinzger 2005] analyzed change-sets from a CVS repository to distinguish changes
within source code entities such as classes and methods (termed as structural changes)
from the changes to license updates and white space between source code entities (termed
as non-structural changes.) The goal of their work was to refine evolutionary couplings
detected from version history with this information (i.e., reduce false positives.) Their
study on an Eclipse plugin found over 31% change-sets with no structural changes and
over 51% change-sets with at least one non-structural change. In one of the rare cases,
Ying et. al. [Ying et al. 2004] defined the interestingness measure of the evolutionary
coupling based on the source code dependencies such as calls, inheritance, and usage.
Their study on Eclipse and Mozilla found evolutionary couplings that were not
represented by the source code dependencies they considered. We feel that further
utilizing such source code dependencies (such as association and dependency
relationships defined in UML) will result in developing heuristics and criteria that would
further reduce false evolutionary couplings. It will also help to detect evolutionary
couplings that are prevalent but do not exhibit any source code dependencies (e.g.,
domain or developer induced dependencies.) More studies in this direction are needed to realize the exclusive and synergistic contributions of MSR approaches.

2.8.2 Historical Context – How many Versions?

Software repositories bring a rich history of software development and evolution. One goal of MSR is to uncover the past successes, and failures, from historical information and improve the evolution process of the software system(s) under consideration. However, one needs to be careful when selecting the amount and period of historical data for basing tools or models supporting a particular aspect of software evolution. Considering the development data too far back in the history imposes a risk of irrelevant information. The design or operational assumptions of the system may no longer be similar, or worse may be entirely different. For example, consider a hypothetical system that has undergone 1,000 versions. The information about the changes in the first 50 versions may be totally irrelevant for predicting the changes in the 1,001st version. A series of changes from version 50 to version 200 could be attributed to an unstable unit in the system that has now stabilized.

On the other hand, considering too few versions of the system imposes the risk of being incomplete or missing important relevant information thus resulting in little useful results. For example, a current version of a system may be in the middle of a refactoring that is achieved by a sequence of changes (versions.) At minimum the past versions beginning from when the refactoring started are needed to first confirm the kind of refactoring taking place and predict the remaining steps. The number of versions to mine depends on the task and the current state/phase of the system under consideration.
2.8.3 Threats to Validity in MSR

MSR approaches use a variety of software repositories, ask different questions, and draw conclusions within the context of the conducted study. All these factors are subject to threats to validity.

Gasser et al. [Gasser, Ripoche, Sandusky 2004] identify the challenges associated with the common need among researchers in selecting, gathering, and maintaining the raw data of open-source projects for their respective investigations. They suggest a research infrastructure to deal with such challenges and to serve as a benchmark to facilitate comparative and collaborative research. They discuss the infrastructure with regards to representation standards for data and metadata available in various software repositories, linking them, the required tools, and a centralized data repository. German further suggests a set of projects representing various sizes and domains, their extracted source code facts (i.e., syntax and semantic), and the period of considered history and observation for these projects to be benchmarked [German 2004b; German, Cubranic, Storey 2005].

We call for a comparative framework to objectively compare MSR approaches with regards to the aspects of software evolution, MSR questions, and the results. Such a framework will facilitate more generic conclusions in the MSR research. Currently, it is difficult to see that two independent MSR investigations are asking equivalent questions or studying the same or similar aspect of software evolution. A benchmark of this nature would help address the expressiveness and effectiveness of MSR in improving software evolution.
2.9 Summary

Over 80 investigations were surveyed that examine multiple snapshots of software artifacts (e.g., source code version from CVS, system release, etc.) and/or other temporal information (e.g., effect on size and structure of a system, bug reports, etc.) From this survey of the literature, a layered taxonomy was derived that characterizes the software repositories utilized, the purpose of the investigation, the methodology used, and the evaluation methods. Each investigation was then categorized within this taxonomy.

The taxonomy facilitates comparison of new approaches/investigations for mining information from software repositories by the research community. Previously, no overarching survey or taxonomy of this literature has been presented. The intent of this work is to form a basis for those researchers interested in mining software repositories for the purpose of understanding the evolution of a software system. Our hope is this taxonomy will assist in the continued advancement of the field.

We feel that the work presented here is a prerequisite to understanding what additional contributions MSR approaches bring to the table for understanding software evolution, beyond that of other software engineering research (e.g., traditional program analysis techniques or software metrics.) A clearer understanding will support the development of tools, methods, and processes that more precisely reflect the actual nature of software evolution.
Table 1. The subset of surveyed approaches that study changes to artifacts.

<table>
<thead>
<tr>
<th>Purpose: MSR Question</th>
<th>Representation</th>
<th>Information Sources</th>
<th>Technique</th>
<th>Task</th>
<th>Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>MBQ &amp; PQ</td>
<td>Files, Classes</td>
<td>Metadata – CVS</td>
<td>Logical Couplings and Change Patterns</td>
<td>Gall [Gall, Hajek, Jazayeri 1998; Gall, Jazayeri, Krajewski 2003]</td>
<td></td>
</tr>
<tr>
<td>MBQ &amp; PQ</td>
<td>Files</td>
<td>Metadata – CVS and Bugzilla</td>
<td>Metadata Analysis</td>
<td>Fischer [Fischer, Pinzger, Gall 2003]</td>
<td></td>
</tr>
<tr>
<td>MBQ &amp; PQ</td>
<td>Projects, Files</td>
<td>Metadata – CVS and Bugzilla</td>
<td>Bug-Fixing Change Analysis</td>
<td>Fischer [Fischer et al. 2005]</td>
<td></td>
</tr>
<tr>
<td>MBQ</td>
<td>Files, Lines</td>
<td>Metadata – CVS and Bugzilla</td>
<td>Characteristics of Different Types of Changes</td>
<td>German [German 2004a]</td>
<td></td>
</tr>
<tr>
<td>MBQ</td>
<td>Files, Comments</td>
<td>Metadata – CVS</td>
<td>Formalism for Querying Metadata</td>
<td>Hindle [Hindle, German 2005]</td>
<td></td>
</tr>
<tr>
<td>MBQ &amp; PQ</td>
<td>Components of a function declaration</td>
<td>Source Code – CVS</td>
<td>Function Interface Changes</td>
<td>Kim [Kim, Whitehead, Bevan 2005]</td>
<td></td>
</tr>
<tr>
<td>MBQ &amp; PQ</td>
<td>Files, Functions, Variables Context: (Change_Type)</td>
<td>Metadata, Source Code – CVS</td>
<td>Itemset Mining</td>
<td>Zimmermann [Zimmermann et al. 2004; Zimmermann et al. 2005]</td>
<td></td>
</tr>
<tr>
<td>MBQ &amp; PQ</td>
<td>Files</td>
<td>Metadata – CVS</td>
<td>Change Prediction</td>
<td>Ying [Ying et al. 2004]</td>
<td></td>
</tr>
<tr>
<td>MBQ</td>
<td>Methods</td>
<td>Metadata, Source Code – CVS</td>
<td>Call-Usage Patterns</td>
<td>Livshits [Livshits, Zimmermann 2005]</td>
<td></td>
</tr>
<tr>
<td>MBQ</td>
<td>Files</td>
<td>Metadata – CVS</td>
<td>Maintenance Relevance Relations</td>
<td>Shirabad [Shirabad, Lethbridge, Matwin 2001; Shirabad, Lethbridge, Matwin 2003; Shirabad, Lethbridge, Matwin 2004]</td>
<td></td>
</tr>
</tbody>
</table>
### Table 2. The subset of surveyed approaches studying changes to properties.

<table>
<thead>
<tr>
<th>Purpose: MSR Question</th>
<th>Representation</th>
<th>Information Sources</th>
<th>Technique</th>
<th>Task</th>
<th>Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>MBQ &amp; PQ</td>
<td>System, Files</td>
<td>Metadata – GNATS, CVS, and Email</td>
<td>Metadata Analysis</td>
<td>Validating Hypotheses for Successful open source development</td>
<td>Dinh-Trong [Dinh-Trong, Biesman 2005]</td>
</tr>
<tr>
<td>MBQ</td>
<td>Files</td>
<td>Metadata – Proprietary Repository</td>
<td>Relationships between Bugs/Features</td>
<td>Sandusky [Sandusky, Gasser, Ripoche 2004]</td>
<td></td>
</tr>
<tr>
<td>PQ</td>
<td>Packages, Files</td>
<td>Versions of a distribution</td>
<td>Bugs Finding and Fixing</td>
<td>Williams [Williams, Hollingsworth 2004; 2005a]</td>
<td></td>
</tr>
<tr>
<td>PQ</td>
<td>Function, Function Calls – Context: (File, Line),</td>
<td>Source Code - CVS</td>
<td>Factors for Successful Software Reuse</td>
<td>Selby [Selby 2005]</td>
<td></td>
</tr>
<tr>
<td>MBQ &amp; PQ</td>
<td>Function Calls, Parameters, Comments, Control and Assignment Statements</td>
<td>Metadata – Manuals Source Code – Proprietary Repository</td>
<td>Function Usage Patterns</td>
<td>Williams [Williams, Hollingsworth 2005b]</td>
<td></td>
</tr>
<tr>
<td>MBQ &amp; PQ</td>
<td>Files, Classes, Methods, Parameters</td>
<td>Source Code – CVS</td>
<td>Communications Via Source Code Comments</td>
<td>Ying [Ying, Wright, Abrams 2005]</td>
<td></td>
</tr>
<tr>
<td>PQ</td>
<td>Comments</td>
<td>Source Code – CVS</td>
<td>System Complexity</td>
<td>Capiluppi [Capiluppi, Morisio, Ramil 2004]</td>
<td></td>
</tr>
<tr>
<td>MBQ</td>
<td>System</td>
<td>Metadata – CVS</td>
<td>Visualization</td>
<td>Van RysSELberghe [Van RysSELberghe, Demeyer 2004b]</td>
<td></td>
</tr>
<tr>
<td>MBQ</td>
<td>Files</td>
<td>Metadata – CVS</td>
<td>Change Smells and Refactoring</td>
<td>Ratzinger [Ratzinger, Fischer, Gall 2005]</td>
<td></td>
</tr>
<tr>
<td>MBQ &amp; PQ</td>
<td>Classes</td>
<td>Metadata – CVS</td>
<td>Clone Detection Methods</td>
<td>Kim [Kim, Notkin 2005]</td>
<td></td>
</tr>
<tr>
<td>MBQ</td>
<td>Files</td>
<td>Source Code – CVS</td>
<td>Clones and their Relationships</td>
<td>Kim [Kim, Notkin 2005]</td>
<td></td>
</tr>
<tr>
<td>MBQ</td>
<td>Files. lines</td>
<td>Metadata – CVS and Bugzilla</td>
<td>Change Prediction</td>
<td>Canfora [Canfora, Cerulo 2005; Canfora, Cerulo 2006]</td>
<td></td>
</tr>
<tr>
<td>MBQ</td>
<td>Variables</td>
<td>Source Code</td>
<td>Concept Keywords for Program Comprehension</td>
<td>Ohba [Ohba, Gondow 2005]</td>
<td></td>
</tr>
<tr>
<td>MBQ</td>
<td>Document Text</td>
<td>Proprietary Repository for Requirement Traceability</td>
<td>Importance of Human Guidance</td>
<td>Hayes [Hayes, Dekhtyar, Sundaram 2005]</td>
<td></td>
</tr>
<tr>
<td>MBQ</td>
<td>Files, Documents</td>
<td>Metadata – CVS, Bugzilla, and Email</td>
<td>New Developer Assistance</td>
<td>Cubranic [Cubranic, Murphy 2003; Cubranic et al. 2005]</td>
<td></td>
</tr>
<tr>
<td>MBQ</td>
<td>Bug Reports</td>
<td>Metadata – Bugzilla</td>
<td>Classification Supervised Learning</td>
<td>Anvik [Anvik, Hiew, Murphy 2006]</td>
<td></td>
</tr>
<tr>
<td>MBQ</td>
<td>Directories</td>
<td>Metadata – CVS</td>
<td>Social Network Analysis</td>
<td>Huang [Huang, Liu 2005]</td>
<td></td>
</tr>
<tr>
<td>MBQ &amp; PQ</td>
<td>Projects</td>
<td>Metadata – Sourceforge</td>
<td>Inter-Project Collaboration</td>
<td>Ohira [Ohira et al. 2005; Ohira et al. 2004]</td>
<td></td>
</tr>
</tbody>
</table>
Table 3. The subset of surveyed approaches studying changes to both artifacts and properties.

<table>
<thead>
<tr>
<th>Purpose: MSR Question</th>
<th>Representation</th>
<th>Information Sources</th>
<th>Technique</th>
<th>Task</th>
<th>Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>MBQ &amp; PQ</td>
<td>Files, Functions, Variables</td>
<td>Metadata, Source Code – CVS</td>
<td>Heuristics for Change Predictions</td>
<td>Hassan [Hassan, Holt 2004]</td>
<td></td>
</tr>
<tr>
<td>MBQ &amp; PQ</td>
<td>File, Functions</td>
<td>Metadata, Source Code – CVS</td>
<td>Software Metrics</td>
<td>Complexity of Different Type of Changes</td>
<td>Nikora [Nikora, Munson 2003]</td>
</tr>
<tr>
<td>MBQ &amp; PQ</td>
<td>System, Functions</td>
<td>Metadata – Proprietary Repository</td>
<td>Types of Changes and Origin Analysis</td>
<td>Tu [Tu, Godfrey 2002]</td>
<td></td>
</tr>
<tr>
<td>MBQ &amp; PQ</td>
<td>Directories, Files, Functions</td>
<td>Source Code – CVS</td>
<td>Visualization</td>
<td>Structural and Architectural Changes</td>
<td>Tu [Tu, Godfrey 2002]</td>
</tr>
<tr>
<td>MBQ &amp; PQ</td>
<td>Subsystems, Modules</td>
<td>Source Code – CVS</td>
<td></td>
<td></td>
<td>Gall [Gall, Jazayeri, Claudio 1999]</td>
</tr>
<tr>
<td>MBQ &amp; PQ</td>
<td>Subsystems, Modules, Programs</td>
<td>Metadata – CVS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MBQ &amp; PQ</td>
<td>Files</td>
<td>Metadata – Proprietary Repository</td>
<td>Information Retrieval</td>
<td>Classification of MRs Based on the Cause of a Change</td>
<td>Mockus [Mockus, Votta 2000]</td>
</tr>
</tbody>
</table>
Table 4. The approaches surveyed and organized by the MSR tasks they address. The tasks are categorized into related groups.

<table>
<thead>
<tr>
<th>Evolutionary Task Category</th>
<th>Approaches</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source Code Differencing</td>
<td>Maletic [Maletic, Collard 2004], Neamtiu [Neamtiu, Foster, Hicks 2005], Raghavan [Raghavan et al. 2004], Sager [Sager et al. 2006]</td>
</tr>
<tr>
<td>Software Reuse</td>
<td>Selby [Selby 2005], Van Rysselberghe [Van Rysselberghe, Demyer 2004a], Xie [Xie, Pei 2006]</td>
</tr>
</tbody>
</table>
Figure 1. Four-Layer Taxonomy of MSR Approaches.
CHAPTER 3

Uncovering Traceability Links from Commits in Software Repositories

Traceability link recovery has been a subject of investigation for many years within the software engineering community [Gotel, Finkelstein 1994; Spanoudakis, Zisman 2005]. Having explicit documented traceability links between various artifacts (e.g., source code and documentation) is vital for a variety of software maintenance tasks including impact analysis, program comprehension, and requirements assurance of high quality systems. It is particularly useful to support source code comprehension if links exist between the source code, design, and requirements documentation. These links help explain why a particular function or class exists in the program.

A number of techniques have been proposed to assist in the recovery/discovery of traceability links in existing software systems [Spanoudakis, Zisman 2005]. However, the problem has proven to be quite difficult and no single approach is, by itself, completely successful or accurate. Many approaches suffer from producing many false positives, suggesting a link when none should exist [Antoniol et al. 2002; Lormans, Van Deursen 2005; Marcus, Maletic 2003].

Approaches to recovering/discovering traceability links typically analyze a single snapshot (i.e., current version) of a software system to infer links between two or more artifacts. The method presented here to uncovering traceability links differs in that it examines multiple versions (i.e., change history) of the software artifacts to recover
traceability links. In [Kagdi, Maletic 2007c] we developed the premise that if artifacts of different types (e.g., src.cpp and help.doc) are co-changed with high frequency over multiple versions, then such artifacts potentially have a traceability link between them.

This history-based approach offers some distinct advantages over single version approaches, namely:

• Traceability links are derived from the actual changes to artifacts, rather than estimations that are based on the analysis of various structural and semantic dependencies between them in a single snapshot of a system.

• The change history represents one of the few sources of information available for recovering traceability links that is manually created and maintained by the actual developers. The commits, and commit messages embody part of the developer’s knowledge and experience. The version history may contain domain-specific “hidden” links that program-analysis methods fail to uncover.

We mine the version history found in software repositories that are maintained by version-control tools such as Subversion or CVS. Specifically, we use the data mining technique sequential-pattern mining [Agrawal, Srikant 1995] to identify and analyze sets of files that are committed together. This technique produces frequently co-changed files, otherwise known as change patterns. Here, these change patterns are considered as a representation of evolutionary couplings among the involved artifacts. Evolutionary couplings that include files representing different types of artifacts are considered related via a traceability link. Another approach, itemset mining [Goethals 2005], has also been used for mining version histories. Itemset mining produces change patterns that are
unordered sets of co-changing artifacts; whereas sequential-pattern mining produces ordered (actually partially ordered) lists of co-changing artifacts. That is, the order in which the artifacts were changed (or committed across multiple revisions) is preserved. The ordering information can thus be used to infer directionality of the traceability links.

Our approach is evaluated on the open-source software system KDE (K Desktop Environment.) The results show that our approach is able to uncover traceability links between various types of software artifacts (e.g., source code files, change logs, user documentation, and build files) with high accuracy. This approach can readily be applied, in conjunction with other link recovery methods, to produce a more complete picture of the traceability of a software system.

3.1 Open Source and Traceability

Scacchi et al. [Scacchi 2002] observed that requirements elicitation, analysis, and specification of open-source system are very different from the traditional approaches (e.g., use of mathematical logic, descriptive schemes, and UML design models) in software engineering. Their requirements are typically implied by discourse of project participants, and after implementation assertions. Different types of informal sources (termed as software informalisms) form collective requirements and documentation of an open-source project. This includes software repositories, communications, HowTo guides, and traditional system documents (e.g., man pages.) One particular type of requirements that is not an uncommon feature in many open-source projects is the ability to support extension mechanisms with various programming languages and architecture (e.g., a python binding to the KDE libraries.) Due to the distributed collaborative nature
of open development, software repositories comprise the primary location of project artifacts along with the primary means of coordination and archival.

The bug/issue tracking repositories and emails can be seen as a source for requirements and corrective-maintenance requests of an open-source system. Source-control repositories can be seen as a source of implementation artifacts. Few efforts have been made to infer and then utilize traceability links between artifacts in bug repositories and source code artifacts via Mining Software Repositories (MSR.) Canfora et al. [Canfora, Cerulo 2005] used the bug descriptions and the CVS commit messages for the purpose of change predictions. Their approach provides a set of files (at line level of granularity) that are likely to change given the textual description of a new bug (or feature.) An information-retrieval method is used to index the changed files in the CVS repositories with the textual description of past bug reports in the Bugzilla repository and the CVS commit messages. A bug report is linked to a CVS commit (i.e., a set of changed files) based on the explicit bug identifier found (a common practice in open-source development) in that commit message (e.g., bug id 30,000.) Sliwerski et al. [Sliwerski, Zimmermann, Zeller 2005] used a combination of information in the CVS log file (commits) and Bugzilla to study fix-inducing changes. Fix-inducing changes are the changes that introduced new changes to fix an earlier reported problem. Regular-expression matching on the commit messages and text descriptions in Bugzilla along with heuristics are used to determine the CVS deltas that are related to a change that fixes a bug. Cubranic et al. [Cubranic et al. 2005] describes a tool, namely Hipikat, to assist new developers (not necessarily novice) on a project, in performing their current task(s.)
Hipikat recommends artifacts from the project memory that may hold relevance to a task at hand. A developer may ask for the relevant artifacts explicitly in the form of an explicit query, or the tool can do so automatically based on the current context (e.g., based on the currently open document(s) in the developer’s workspace.)

In summary, existing MSR approaches have focused on uncovering traceability links between requests in bug-tracking systems and source code. While these are important efforts, they cover only a portion of the broad spectrum of documents found in open-source development. A sustainable success of an open-source project, from both development and end use perspectives, depends to a large extent on how well they maintain these documents. For example, an application that frequently fails to compile or with very little installation help could have a diminishing effect on the user base. It is important that these documents be kept in alignment with the current state of the source code. Therefore, traceability between them is of desirable interest and value. Accounting these documents along with the requests in bug-tracking systems is a major step towards achieving the complete picture of traceability to source code in the context of open-source development.

3.2 Uncovering Traceability Links

Our research interest is in uncovering traceability between source code and other artifacts. This includes user documents (e.g., HTML, XML/docbook, LaTeX and Doxygen), build management documents (automake, cmake, and makefile), HowTo guides (e.g., FAQs), release and distribution documents (e.g., ChangeLogs, whatsNew, README, and INSTALL guides), progress monitoring documents (TODO and STATUS),
and extensible mechanisms (e.g., Python, Ruby, and Perl bindings for an API.) These artifacts can be considered software informalisms [Scacchi 2002]

Our approach is to analyze ordered sets of files that frequently co-occur in change-sets by applying a frequent-pattern mining technique (i.e., sequential pattern mining.) We refer to such a list of ordered sets of files as an evolutionary coupling. These evolutionary couplings are then analyzed to uncover patterns that contain source code files and other types of files. We refer to such a pattern as a traceability coupling. Therefore, here traceability patterns/couplings are considered as the manifestation of traceability links.

Our hypothesis is that if the same set of files that are of different types, co-change with a high frequency then there is a potential traceability link between them. For this to be a sound hypothesis, the basic prerequisite is to examine if different types of files are typically changed together in the first place. Our study on six open source systems [Kagdi, Maletic 2007c] shows that between 28% and 62% of the change-sets (i.e., a set of files checked into a software repository together in a single commit operation) contain two or more types of artifacts. These systems cover a number of application domains, sizes, and are primarily written in C, C++, and Java. Apache httpd is a web server, jEdit is an editor, GCC is a compiler, koffice is an office-applications suite, kdelibs is a core library for KDE (K Desktop Environment), and Python is a programming language. We believe such change-sets are a valuable source for uncovering traceability links. Using this source, an approach that discovers a specific set of files with traceability links between them can be devised. We first describe how change-sets are stored and
represented in software repositories to help facilitate the following discussion of our mining approach for traceability links.

In order to establish whether different types of documents are committed in the same change-set six open source systems are examined. These systems cover a number of application domains and are primarily written in C, C++, and Java. Apache httpd is a web server, jEdit is an editor, GCC is a compiler, koffice is an office-applications suite, kdelibs is a core library for KDE (K Desktop Environment), and Python is a programming language. All these projects use Subversion for managing their repositories. Change-sets committed in periods between one and six years were considered. In some cases only recent history of about a year and half was considered to mitigate the influence of “old” changes that may be irrelevant for the current state of the system and its further evolution. We selected 10% of these change-sets via random sampling. These sampled change-sets were analyzed for the number of different types of artifacts in them.

Figure 2 shows the frequency distribution of the number of different file types with regards to the number of change-sets in the change-set samples. Our analysis indicates that a substantial proportion of change-sets contain two or more file types in these systems. Change-sets with a fewer number of different file types occur more frequently than those with a larger number of different file types.
Table 5 provides some descriptive statistics of the sampled change-sets. The proportions of the change-sets with two or more file types (column Proportion) are between 28% and 62%. The sample means (column Mean) and standard deviations (column SD) are also given. Both these measures show that on average a change-set contains more than one file type, and as high as three file types. The standard deviations indicate that change-sets with a number of different file types also appear. We performed an outlier analysis via Inter Quartile Range (IQR) computation to determine the change-sets that deviate from a typically “normal” case (an outlier.) That is, they contain a large number of different types than what is typically observed. The limits on the number of different types in a change-set beyond which it can be considered as a suspect for an outlier are also determined from the samples (Column Outlier Cut-off.) This analysis suggests that a change-set with the maximum range [3, 6] can well be the “normal” case.
To give an indication as to how well the statistics on the samples represent parametric means of all the change-set (i.e., within and beyond the history period considered for selecting the samples) in these projects, we provide the confidence intervals (column CI) for estimating the overall means. The confidence intervals were computed with the confidence level of 95%. That is, we can say with a 95% confidence that the mean of all the change-sets in a project will be within the given bounds. As can be seen, the bounds do not vary much the overall mean from the sample means for all the six systems.

The analysis presented in this section shows that there are change-sets with more than one document type in software repositories. Utilizing this information to infer potential traceability link between them is a two-fold issue: 1) Is the presence of different types of documents in the same (and single) commit enough to infer the traceability links between them? 2) How do we account for related documents with potential traceability links committed in a series of multiple change-sets? We now present a heuristic-based approach that uses data mining methodology that addresses theses issues.

Table 5. Statistics of the change-sets analyzed for different types of files in six open source systems

<table>
<thead>
<tr>
<th>System</th>
<th>Proportion</th>
<th>Mean</th>
<th>SD</th>
<th>Outlier Cut-off</th>
<th>CI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache-httpd</td>
<td>34%</td>
<td>1.821</td>
<td>3.0858</td>
<td>3.5</td>
<td>±0.6599</td>
</tr>
<tr>
<td>GCC</td>
<td>32%</td>
<td>2.532</td>
<td>1.6547</td>
<td>4.5</td>
<td>±0.0882</td>
</tr>
<tr>
<td>jEdit</td>
<td>62%</td>
<td>2.212</td>
<td>1.3489</td>
<td>6.0</td>
<td>±0.1833</td>
</tr>
<tr>
<td>kdelibs</td>
<td>33%</td>
<td>1.487</td>
<td>1.0689</td>
<td>3.5</td>
<td>±0.0652</td>
</tr>
<tr>
<td>koffice</td>
<td>45%</td>
<td>1.793</td>
<td>1.6448</td>
<td>3.5</td>
<td>±0.0934</td>
</tr>
<tr>
<td>Python</td>
<td>28%</td>
<td>1.441</td>
<td>0.9065</td>
<td>3.5</td>
<td>±0.0933</td>
</tr>
</tbody>
</table>
3.2.1 Change-sets in Software Repositories

Source code repositories store metadata such as user-IDs, timestamps, and commit comments in addition to the source code artifacts and their differences across versions. This metadata explains the why, who, and when dimensions of a source code change. Modern source-control systems, such as Subversion, preserve the grouping of several changes in multiple files to a single change-set as performed by a committer. 

Version-number assignment and metadata are associated at the change-set level and recorded as a log entry.

Figure 3 shows a log entry from the Subversion repository of kdelibs (a part of KDE repository.) A log entry corresponds to a single commit operation. This information can be readily obtained in an XML format by using the command-line client svn log. Subversion’s log entries include the dimensions author, date, and paths involved in a change-set. In this case, the changes in the files khtml_part.cpp and loader.h are committed together by the developer kling on the date/time 2005-07-25T17:46:20.434104Z. The revision number 438663 is assigned to the entire change-set (and not to each file that is changed as is in the case with some version-control systems such as CVS.) Additionally, a text message describing the change entered by the developer is also recorded. Note that the order in which the files appear in the log entry is not necessarily the order in which they were changed. Clearly, a single log entry alone is insufficient to give the temporal ordering in which files were changed. However, there is a temporal order between change-sets. Change-sets with greater revision numbers occur after those with lesser revision numbers. Therefore, we can utilize the ordering of
change-sets to determine the ordering of changes between different files. In the rest of the chapter, we use the term change-sets for the log entries in Subversion repositories. Let us now describe relevant data-mining terminology and the frequent-pattern mining techniques.

![Figure 3. A Snippet of kdelibs Subversion Log](image)

### 3.2.2 Data Mining Background

The input data to frequent-pattern mining algorithms are in the form of transactions. A transaction refers to a group of items that share a common property or occur in the same event (e.g., customer baskets or items checked-out together in market-basket analysis.) The number of transactions in which a pattern occurs is known as its support. The support of a pattern is the number of groups in which it appears. So if the support of a pattern is at least a user-specified minimum support then it is a frequent pattern in the considered dataset. Sequential-pattern mining produces a partially ordered list of files for patterns and as such we term these ordered patterns.

Sequential-pattern mining takes a set of sequences and finds all the frequently occurring subsequences (i.e., ordered patterns) that have at least a user-specified
minimum support [Masseglia, Teisseire, Poncelet 2005]. Here, transactions are in the form of sequences of items. Sequential-pattern mining techniques are typically applied to datasets with temporal or other ordering information. For example, in case of market-basket analysis with the additional timestamp information, sequential patterns such as customers who bought a camera are also likely to buy additional memory in the next month.

### 3.2.3 Mining Ordered Patterns

Software is inherently structured with dependencies among its entities such as call, control, and data dependencies. The task of performing a software change is either planned (e.g., a standard refactoring or a fix for a documented bug), unplanned activity (e.g., fixing an unforeseen side effect due to a change), or a combination of both. A typical planned change is implemented in small increments with the goal of maintaining the overall system in a coherent state (e.g., preserve the build or compile-able state, change source code and documentation in separate steps.) These incremental changes corresponding to the change-sets are implicitly ordered. However, such is the nature of software that an extremely well planned change may lead to further unanticipated changes. It is not uncommon to have a bug-fix that introduces a multitude of additional bugs. Often such bugs are discovered only after a fix is committed to a repository and tested by (possibly other) developers or users. Nonetheless, in any case there is a temporal ordering between various change-sets in the repository.

Preservation of a change-set as an atomic commit in software repository gives the ability to iterate through the change history at the change-set level (i.e., “undo” at the
change-set level rather than the individual file level.) This encourages the practice of committing a set of related changes in a single logical change – a standard Subversion policy of the KDE project. However, the granularity and composition of a change-set may vary across tasks, developers, and projects. For example, consider a refactoring task that requires a series of steps such as extract method, move method, and so forth. A change-set may correspond to each elementary step or the entire refactoring. In other cases, changes to source code and related documents may be committed at different times even though they represent the same logical change. Therefore, a single high-level change may be completed over multiple change-sets.

In order to mine larger or more complete patterns we need to consider changes that spread over a sequence of change-sets. However, the changes-sets corresponding to such changes are rarely explicit (at least not directly recorded in the software repositories or clearly documented.) Notice that the change-sets stored as atomic commits in software repositories are serialized. The order in which log entries appear in the log files is at the discretion of a version-control system. Two unrelated change-sets committed approximately at the same time may appear next to each other. Therefore, treating successive change-sets in the software repositories as related to a single high-level change may be meaningless.

In our approach we use three heuristics to group change-sets. Each heuristic takes a set of change-sets and forms groups of “related” change-sets. From the discussion in section 3.2.1, there is a temporal relationship between change-sets. Therefore, each group formed by heuristics is actually a sequence of change-sets.
We employ sequential-pattern mining to uncover ordered evolutionary couplings from the groups formed by a grouping heuristic. The transactions are the groups (i.e., sequence of change-sets) and the items are the files. The ordered patterns discovered by sequential-pattern mining are the sequences of files (actually a sequence of sets of files) that are found common in at least a user-specified number of groups (i.e., minimum support.)

In general an ordered pattern is composed of elements. Each element is composed of unordered items. The ordering of elements imposes a partial order on the items. For example, the ordered pattern \{f_1, f_2\}→\{f_3, f_4\}→\{f_5\} is composed of three elements and five items. It indicates that the element \{f_1, f_2\} happens before the element \{f_3, f_4\} and the element \{f_3, f_4\} happens before the element \{f_5\}. However, the happens before relation between items \(f_3\) and \(f_4\) is unknown in the element \{f_3, f_4\}. In the context of ordered evolutionary couplings, an element in an ordered pattern corresponds to a subset of files changed in a change-set and an item in an element corresponds to a file. Therefore, files in the same element of an ordered pattern indicate files that are likely to change in the same change-set, whereas files in the different elements of an ordered pattern indicate files that are likely to change in different change-sets in the specified order. For the sake of brevity, ordered evolutionary couplings are referred as ordered patterns in the remainder of this discussion.

The support of an ordered pattern is the number of groups in which it occurs. An ordered pattern indicates that if any of its constituent files are found in a change-set then the rest of the files are also likely to occur in the same or different change-set as per their
ordering in the pattern. Therefore, an ordered pattern in the context of a software repository could mean a set of files that are likely to be committed in the same revision before a set of files committed in the previous revision.

3.2.4 Change-set Grouping Heuristics

We present three heuristics for grouping related change-sets formed from version history metadata found in software repositories (i.e., developer, time, and changed files.) These heuristics can be considered similar to the fixed and sliding window techniques [Gall, Hajek, Jazayeri 1998; German 2004b; Zimmermann et al. 2005]. These techniques are used to group changed files into a single change-set typically applied to CVS repositories as they lose the atomicity of original change-sets (unlike change-sets in Subversion.) Our heuristics combine change-sets into groups in order to account for related changes committed across multiple change-sets.

Time Interval.

This grouping heuristic is based on the premise that the change-sets committed during a given time-interval are related, and change-sets committed outside this interval are unrelated. All the change-sets committed in a given time duration are placed in a single group. The number of groups is equal to the number of unique time intervals over which the change-sets were committed. This heuristic covers related change-sets that are committed by different developers but during the same time interval. The ordered patterns found using this heuristic implies that if a file is modified in a particular pattern
within a given time interval, the following (or preceding) files are likely to be modified on the same day.

For example, the pattern \{khtml_part.h\} → \{ChangeLog\} was found from mining the change-sets in the KDE Subversion repository (under kdelibs/khtml/) committed between May 2005 and December 2005. In this case, a group was formed for the change-sets committed in one calendar day. This pattern is found to occur in five groups. On each of these five days, the file khtml_part.h was in a change-set that was committed before the change-set in which the file ChangeLog was committed. This is a traceability coupling showing that changes are documented after an interface file is changed. The pattern

\{kdeedu/kalzium/src/kalzium.cpp, kdeedu/kalzium/src/pse.cpp\} → \{kdesdk/doc/scripts/kdesvn-build/index.docbook\}

is another example pattern that occur in change-sets committed in each of five different days. This pattern shows that the documentation is updated after performing changes to the source code. However, the order in which the two source code files were changed cannot be determined (i.e., a partially ordered pattern.)

Committer.

This heuristic is based on the premise that the change-sets committed by a single developer are related and the change-sets committed by different committers are unrelated. This defines an order on the change-sets by a committer. Therefore, all the change-sets committed by a given committer are placed in a single group.
The number of groups is equal to the number of unique committers. This heuristic covers related change-sets that are committed in different time intervals but by the same author. The ordered pattern found using this heuristic implies that if a file is modified in a pattern by a committer, the following or preceding files are likely to be modified by the same committer.

The pattern \{khtml\_part\_h\}→\{ChangeLog\} was found from mining the change-sets in the KDE Subversion repository committed between May 2005 and December 2005. A group in this case was formed for the change-sets committed by the same developer. This pattern is found to occur in five groups. In the case of each committer, the file kdelibs/khtml/khtml\_part\_h was in a change-set that was committed before the change-set in which the file kdelibs/khtml/ChangeLog was committed. The same pattern was found by grouping change-sets by the heuristic Time interval (see section 0.) This further strengthens that this is a change dependency between these artifacts and not an unrelated dependency due to a development practice of a developer or unusual changes made during a particular day. The pattern

\{kdeedu/kalzium/src/kalziumentip.cpp\} → \{kdeedu/kalzium/src/detailinfodlg.cpp\} → \\
\{kdeedu/kalzium/src/Makefile.am\} → \{kdeedu/kalzium/src/kalzium.cpp, kdeedu/kalzium/src/kalzium.h\}

is another example pattern that is found in the change-sets committed by five developers. This pattern shows that a build file is updated both before and after changing the source code.
This heuristic is based on the premise that the change-sets committed by the same committer within a time interval are related, and the change-sets committed by the same or different committers in different time intervals are unrelated. This defines an order on the change-sets by a committer. Therefore, all the change-sets committed by a given committer within the same time interval are placed in a single group. The number of groups is equal to the number of unique committers and time interval combinations. This heuristic restricts related change-sets to the change-sets committed by an author within a specific time period. The ordered pattern found using this heuristic implies that if a file is modified in a pattern by a committer the following or preceding files are likely to be modified by the same committer in the same time interval.

For example, the pattern \{TODO\}→\{pse.cpp\} was found from mining the change-sets in the KDE Subversion repository committed between May 2005 and December 2005. A group in this case was formed for the change-sets committed by the same developer on the same calendar day. This pattern is found to occur in ten groups. In each combination of committer and day, the file kdeedu/kalzium/TODO was in a change-set that was committed before the change-set in which the file kdeedu/kalzium/src/pse.cpp was committed. The pattern

\{kdeedu/kalzium/src/kalziumui.rc\} → \{kdeedu/kalzium/src/pse.h,

kdeedu/kalzium/src/pse.cpp\}
is another example pattern that is found in the change-sets committed by seven different committer-day combination. This pattern shows that a particular user-interface file is changed before modifying the code.

3.2.5 Frequent-Pattern Mining Tool

We have developed a sequential-pattern mining tool, namely sqminer, that is based on the Sequential Pattern Discovery Algorithm (SPADE) [Zaki 2001] which utilizes an efficient enumeration of ordered patterns based on common-prefix subsequences and division of search space using equivalence classes. Additionally, it utilizes a vertical input-transaction format (i.e., a set of transactions for each file vs. a set of transactions consisting of files) for efficiency.

To help prune the number of candidate patterns produced by the mining techniques, patterns with redundant information are eliminated. A pattern that is frequent means that all possible patterns formed from the subsets of its files are also frequent. The support of a pattern is always less than or equal to the subset patterns. A common pruning mechanism used in frequent-pattern mining is to eliminate all the subset patterns that have the same support of the corresponding larger pattern. Such subset patterns are only used with other larger patterns and not in isolation by themselves. Therefore, they give redundant information that may be of very little meaning. As a result, only disjoint patterns (i.e., patterns with no common files) that subsume all subsets of patterns with the same or higher support are retained. Such patterns are known as closed patterns. Our tool produces only closed patterns.
Frequent-pattern mining algorithms typically report the support of a pattern but not the transactions in which it occurs. Our tool records the transactions in which a pattern is found. For uncovering both unordered and ordered evolutionary couplings, we use the same underlying mining algorithm. The tool sqminer can also be used for frequent itemset mining. In this case the transactions are formed with no ordering information of items. The configuration parameters of sqminer include support, maximum number of items in a pattern, mining of sequence (association) rules, and output in both a flat-file and XML format. For further detail on the XML output format of the ordered patterns and rules, we refer to [Kagdi, Yusuf, Maletic 2006].

3.3 Evaluation

To evaluate our approach to recovering traceability links we use the open-source system KDE (www.kde.org.) KDE has over 4 million LOC. The KDE repository (websvn.kde.org/trunk/KDE) houses around twenty different modules, each containing multiple applications and libraries. The applications in KDE represent a wide spectrum of domains, programming languages, size, and developers.

The evaluation methodology is to first mine a portion of the version history for traceability couplings. We call this the training-set. Next we mine a later part of the version history (called the evaluation-set) and see if the results generated from the training-set can accurately predict changes that occur in the evaluation-set.

We considered the change-sets committed in a twelve-month period in the KDE repository from 2005-05-01 to 2006-04-30. KDE migrated from CVS to Subversion around May 2005 and this was our primary reason for picking this particular time frame.
We allocated $\frac{2}{3}$ of this version history to the training-set: an eight-month period of history starting at 2005-05-01 and ending on 2005-12-26. This training-set contains 14,939 change-sets consisting of 13,037 files. The remainder was allocated to the evaluation-set: a four-month period of history starting at 2005-12-27 and ending at 2006-04-30. This evaluation-set contains 9,008 revisions (i.e., change sets) consisting of 9,070 files. Only change-sets that consisted of ten files or less are considered. This avoids change-sets such as updating the license information on every file or performing merging and copying.

First we need to extract relevant information from the KDE repository. A straightforward approach to extract the log entries from a Subversion repository is to use the client command `svn log` from a working copy of the repository. This approach is not feasible for use-cases in which only the logs stored in software repositories are needed and not the contents of the committed documents. We developed the tool `changeextractor` that uses `pysvn` (a Subversion module for Python) to extract changesets, without using a working copy, from the repository. `changeextractor` takes a repository URL, a start date, and an end date of a history, and extracts the change-sets from the repository logs for a specified period.

The change-sets extracted from the repository are then grouped into sequences according to the grouping heuristics by the tool `groupchanges` (another Python script.) We choose a calendar day as the time interval for the heuristic `Time Interval`. A committer that contributed a change is mapped to a group for the heuristic `Committer`. 
The number of groups and the total number of change-sets involved in these groups for the three heuristics are shown in Table 6.

<table>
<thead>
<tr>
<th>Heuristics</th>
<th>Groups</th>
</tr>
</thead>
<tbody>
<tr>
<td>Day</td>
<td>237</td>
</tr>
<tr>
<td>Committer</td>
<td>330</td>
</tr>
<tr>
<td>CommitterDay</td>
<td>4,884</td>
</tr>
</tbody>
</table>

Table 6. Groups formed from the change-sets extracted from the KDE repository by the different heuristics.

<table>
<thead>
<tr>
<th>Heuristics</th>
<th>EC</th>
<th>TC</th>
<th>TC/EC%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Day</td>
<td>5,839</td>
<td>1,851</td>
<td>37.10</td>
</tr>
<tr>
<td>Committer</td>
<td>718</td>
<td>54</td>
<td>7.52</td>
</tr>
<tr>
<td>CommitterDay</td>
<td>2,372</td>
<td>277</td>
<td>11.68</td>
</tr>
</tbody>
</table>

Table 7. Evolutionary couplings (EC) and traceability couplings (TC) uncovered from the KDE repository. All patterns have a minimum support of five.

3.3.1 Uncovering Traceability Couplings

The groups constructed by the tool groupchanges are fed to the sequential-pattern mining tool sqminer. Mining frequent ordered patterns with sqminer produces a set of closed evolutionary couplings. We configured sqminer to mine patterns with a minimum support of five for all our defined heuristics.

Table 7 shows the uncovered evolutionary couplings and the traceability couplings uncovered by the heuristics. The traceability coupling consists of at least one source code file and at least one file of another type. C++ is the primary programming language for KDE. The files with extensions {.h, .cpp, .cc, .c, .hxx, .cxx } were considered to be containing source code, whereas those that are not source code files are considered as other artifacts (e.g., with extensions .docbook, .xml, and .html, and
We do not restrict other artifacts to a particular set of types; rather we
discover them.

The heuristic Committer uncovered the minimum number of evolutionary
couplings, traceability couplings, and percentage of the traceability couplings in the
evolutionary couplings. The heuristic Day uncovered the maximum number of change
patterns, traceability couplings, and percentage of the traceability couplings.

The traceability couplings mined are not restricted to binary patterns. Table 8
shows the minimum, maximum, and average number of files in the traceability couplings
uncovered from the training-set. Traceability couplings with as many as seven files were
uncovered with the heuristic CommitterDay. Now that our approach is able to find
potential traceability couplings, a measure of the “goodness” is needed to evaluate the
traceability couplings. We validate our approach with three metrics.

<table>
<thead>
<tr>
<th>Heuristic Queries</th>
<th>Number of files</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Min</td>
</tr>
<tr>
<td>Day</td>
<td>2</td>
</tr>
<tr>
<td>Committer</td>
<td>2</td>
</tr>
<tr>
<td>CommitterDay</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 8. The minimum, maximum, and average number of files in the traceability
couplings in the training-set

3.3.2 Validation Metrics

We refer to the traceability couplings in Table 7 as the training-set and the
traceability couplings in Table 9 as the evaluation-set. The usefulness of the uncovered
traceability couplings can be seen in “how well” the training-set predicts the existence of
a traceability coupling in the evaluation-set. We validate this using the metrics coverage, recall, and precision.

Let $T = \{ts_1, ts_2, ..., ts_m\}$ and $E = \{es_1, es_2, ..., es_n\}$ be the training-set and evaluation-set respectively. Consider the pattern $es_i = \{f_1\} \rightarrow \{f_2\} \rightarrow ..., \rightarrow \{f_k\}$ in the evaluation-set that is undergoing changes. To eventually predict this pattern, the training-set can be queried for candidates after each element $\{f_j\}$ of the pattern $es_i$ is changed (or planned to be changed.) Let $Ces_i = Ces_{i1} \cup Ces_{i2} \cup ... \cup Ces_{ik}$ where $Ces_{ij} = \{ts_1, ts_p\}$ be a set of candidate patterns suggested from the training-set after changing the $j^{th}$ element (and previous elements) of the pattern $es_i$.

**Definition:** Covered pattern is a traceability-pattern in the evaluation-set for which there is at least one candidate pattern suggested from the training-set,

$$Covered Patterns = |\{ \forall es_i \in E \Rightarrow |Ces_i| > 0 \}|$$

**Definition:** Coverage is the percentage of the total number of covered patterns to the total number of patterns in the evaluation-set,

$$Coverage = \frac{CoveredPatterns}{|E|} \times 100\%$$

**Definition:** Correctly covered pattern is a covered pattern with at least one suggested candidate pattern from the training-set that is the same (completely identical) or its sub-pattern (partially identical.)

**Definition:** Recall is the percentage of the total number of correctly covered patterns to the total number of patterns in the evaluation-set.

$$Recall = \frac{CorrectlyCoveredPatterns}{|E|} \times 100\%$$
Coverage and recall are indicative of the completeness of the training-set in predicting the evaluation-set. Coverage describes how many traceability couplings, a developer can expect to be recommended from the patterns mined in the training-set. Recall describes how many of these recommendations are “correct”. Ideally, both coverage and recall should be 100% (all patterns in the evaluation-set are correctly predicted in training-set.)

Coverage and recall give only one measure of usefulness of the traceability couplings for software-change prediction. An arguably more important measure is how many total candidate patterns, both correct and incorrect, are suggested from the training-set that require examination for a covered pattern in the evaluation-set.

Definition: Relevant patterns of a covered pattern are the number of correctly covered patterns suggested after a change in its given element.

For example let $es_i = \{f_1\} \rightarrow \{f_2\} \rightarrow \{f_3\}$ be a covered pattern. If the candidate patterns suggested from the training-set are $\{f_1\} \rightarrow \{f_2\} \rightarrow \{f_3\}, \{f_1\} \rightarrow \{f_2\} \rightarrow \{f_4\},$ and $\{f_1\} \rightarrow \{f_5\} \rightarrow \{f_6\}$ after a change in file $\{f_1\}$ in $es_i$, the relevant patterns are two (out of the three.) After a change in the file $\{f_2\}$ in $es_i$ (i.e., $\{f_1\} \rightarrow \{f_2\}$) the suggested candidate patterns are $\{f_1\} \rightarrow \{f_2\} \rightarrow \{f_3\}$ and $\{f_1\} \rightarrow \{f_2\} \rightarrow \{f_4\}$. The relevant pattern is one (out of two.)

Definition: The relevance ratio of a covered pattern is the sum of the ratios of the number of relevant patterns over the number of suggested candidates of all its elements. The relevance ratio in our example is $2/3 + 1/2 + 1 = 2.167.$

$$Relevance \ Ratio \ (es_i) = \sum \frac{relevantPatterns}{|Ces_i|}$$
**Definition:** Precision of a covered pattern is the percentage of relevance ratio weighted over its number of elements. Let $|es_i|$ be the number of elements in $es_i$.

\[
\text{Precision (} es_i \text{)} = \frac{\text{relevanceRatio}}{|es_i|} \times 100\%
\]

Precision of our example is $(2.167/3) \times 100 = 72\%$. In the best case, for any given covered pattern in the evaluation-set, only that pattern is suggested from the training-set after changes to any of its elements (i.e., precision is 100\%). Using these metrics, we can evaluate our approach on the evaluation-set of our KDE study.

<table>
<thead>
<tr>
<th>Heuristics</th>
<th>EC</th>
<th>TC</th>
<th>TC/EC%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Day</td>
<td>1112</td>
<td>143</td>
<td>12.86</td>
</tr>
<tr>
<td>Committer</td>
<td>304</td>
<td>26</td>
<td>8.55</td>
</tr>
<tr>
<td>CommitterDay</td>
<td>835</td>
<td>8</td>
<td>0.9</td>
</tr>
</tbody>
</table>

Table 9. Traceability couplings (TC) from the KDE repository that are used as evaluation data-set for the traceability couplings (TC) in Table 7. All patterns have a minimum support of five.

<table>
<thead>
<tr>
<th>Heuristic Queries</th>
<th>Coverage (%)</th>
<th>Recall (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Day</td>
<td>44</td>
<td>8</td>
</tr>
<tr>
<td>Committer</td>
<td>57</td>
<td>11</td>
</tr>
<tr>
<td>CommitterDay</td>
<td>25</td>
<td>25</td>
</tr>
</tbody>
</table>

Table 10. Coverage and Recall for the evaluation-set (Table 7) compared against the training-set (Table 9.)

3.3.3 Predicting Changes in the Evaluation-set

We use the traceability couplings in the training-set for software-change prediction on the evaluation-set. A similar process to that of analyzing the training-set was followed in mining traceability couplings from the evaluation-set. Table 9 shows the uncovered traceability couplings from the evaluation-set. A total of 177 traceability
couplings were uncovered in the evaluation-set. We evaluated all these patterns with the patterns in training-set with regards to coverage, recall, and precision. The traceability couplings mined with our heuristics can be considered as representing the following types of software-change prediction questions (i.e., the complete change due to prediction will eventually form a traceability coupling):

Day – If the file \( f \) is changed on the day \( d \), what other files are likely to change on the day \( d \)?

Committer – If the developer \( c \) changes the file \( f \), what other files are likely to be change by \( c \)?

CommitterDay – If the developer \( c \) changes the file \( f_1 \) on the day \( d \), what other files will \( c \) likely change along with the file \( f_1 \) on the same day \( d \)?

Table 10 shows the coverage and recall of the traceability couplings in the evaluation-set from the traceability couplings in the training-set. The results show that the heuristic Committer provides the maximum coverage, however, at the cost of a lower recall. Similarly, the heuristic Day provides a high coverage but also at the cost of a lower recall. The heuristic CommitterDay provides the highest recall, however, at the cost of a very low coverage.

Table 11 shows the precision of the traceability couplings in the evaluation-set from the traceability couplings in the training-set. Only the top ten frequent patterns are recommended as candidates for an element change. The minimum, maximum, and average precision of all the traceability couplings are reported. Notice that precision is measured per pattern; coverage and recall are measured for the entire evaluation-set. The
results show that the heuristic *CommitterDay* is likely to provide higher precision than the others.

The overall results show that there is no single heuristic that outperforms others in terms of coverage, recall, and precision. The heuristic *CommitterDay* is likely to produce better recall and precision. It is safe to say that any predictions made by this approach about the existence of traceability links are quite precise. The heuristics *Day* and *Committer* overall provide reasonable coverage and precision, but recall is low.

<table>
<thead>
<tr>
<th>Heuristic Queries</th>
<th>Precision (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Min</td>
</tr>
<tr>
<td>Day</td>
<td>50</td>
</tr>
<tr>
<td>Committer</td>
<td>55</td>
</tr>
<tr>
<td>CommitterDay</td>
<td>100</td>
</tr>
</tbody>
</table>

Table 11. Precision for the evaluation-set (Table 7) compared against the training-set (Table 9.)

3.3.4 Threats to Validity

With regards to internal validity, we are able to uncover traceability couplings from the change-sets committed in the *KDE* repository in various periods of time and sizes (a minimum of seven days) between 2005/05 and 2006/08. We also show our approach effective in recovering traceability links with only a few days (versions) of change history.

Our results are obtained from the change history of over twenty packages in the *KDE* repository. We also applied our approach to Apache *httpd* software repository and found similar results. Both systems are prime examples of successful open-source development, representing different domains and sizes but we do not claim that our results would generalize to any given system (e.g., commercially developed software.)
We believe that our approach is applicable to any software development with a practice of committing related files together in a software repository. Also, our validation results are within the context of only the frequent patterns that reoccur, and not for every change, in the later part of the history.

3.4 Related Work

There are two distinct areas of research that are directly related to our work, namely Mining Software Repositories (MSR) and traceability link recovery.

Kagdi et al. [Kagdi, Collard, Maletic 2007a] present a survey of MSR approaches in the context of software evolution. Our work closely relates to the works by Zimmermann et al. [Zimmermann et al. 2005] and Canfora et al. [Canfora, Cerulo 2005], however with important distinctions. Zimmermann et al. [Zimmermann et al. 2005] mainly focused on uncovering source-code-to-source-code change dependencies using itemset mining. They considered only the source code entities committed together in a single change-set (approximated via sliding-window technique.) Our focus is on uncovering the traceability links between source code and other types of artifacts (note that we also uncover source-code-to-source-code change dependencies.) We also consider files committed over a sequence of change-sets and not just in a single change-set. We use sequential-pattern mining to uncover the ordering information of committed files. Yang et al [Ying et al. 2004] used a similar technique as Zimmermann et al. [Zimmermann et al. 2005] for identifying files that frequently change together. Canfora et al. [Canfora, Cerulo 2005] work is based on the textual similarity of different bug reports (and commit messages) in the change history. An information-retrieval method is
used to index the changed files in the CVS repositories with the textual description of past bug reports in the Bugzilla repository and the CVS commit messages. Our work is based on a common set of files that is changed multiple times in the change history. As such, their work is dependent on the “quality” of the textual description. Additionally, they are only able to find traceability between bugs/features and source code files for those bugs/features entered in the bug-tracking system. Our approach can operate without this information.

Spanoudakis and Zisman [Spanoudakis, Zisman 2005] conducted a comprehensive study of various methods for link recovery that utilize such things as information retrieval, test-cases, and design patterns. Marcus and Maletic [Marcus, Maletic 2003] used Latent Semantic Indexing (LSI) to recover links, with better precision, from documentation to source code on the same set of case studies done by Antoniol et al. [Antoniol et al. 2002] using vector space IR models. A number of other researchers [DeLucia et al. 2006; Hayes, Dekhtyar, Osborne 2003; Lormans, Van Deursen 2005; Settimi et al. 2004; Sundaram, Hayes, Dekhtyar 2005] have also applied IR methods for traceability link recovery. The results of these studies demonstrate the usefulness of IR methods for link recovery. These approaches do not consider, or depend on, multiple versions of a software system to construct the links. In one of the rare studies that examined two versions at a time, Antoniol et al. [Antoniol, Canfora, Lucia 1999] establish traceability links between software releases of an object oriented system to determine inconsistencies.

3.5 Summary

We used versions history to uncover traceability couplings consisting of source code files and other artifacts. A heuristic based approach that uses frequent-pattern mining is presented. The recovered patterns give the specific order in which the files in a pattern were changes. This ordering information can be utilized to infer the directionality (i.e., change causality) of the traceability link (e.g., source→documentation.) We showed that these traceability links support software-change prediction with high precision, if a similar pattern frequently occurred in past versions.

Our work compounded with the existing approaches in uncovering traceability between requests in bug repositories and source code expands the horizons of traceability research via mining software repositories and overall generally. While the discussion here may seem restricted to the open-source development, we believe that our approach is equally applicable in any other development methodology that exhibits different types of
artifacts in the same change-sets. We feel that the choice of the appropriate duration of history that maximizes accuracy for software-change prediction remains an issue of future investigation. Additional heuristics for grouping related change-sets such as textual similarity of commit messages are also being investigated. We feel that mining at a finer granularity for patterns (e.g., class or method level to paragraph) would produce better results; at least in terms of better context to developers. We are developing tools in this direction. Further, we plan to integrate our traceability tools directly into a version-control system.
CHAPTER 4

Mining Fine-Grained Evolutionary Couplings from Source Code Changes

An evolving software system undergoes changes due to defect corrections, feature additions, and design improvements. Irrespective of a specific cause, changes may not always be localized to a single element. Such changes may result in undesirable side effects and/or ripple effects. The end goal is not to leave a system in a state that is erroneous or violates the original assumptions and semantics due to an introduced change.

Maintainers are routinely faced with the common question: If a specific source code entity undergoes change, what other entities would also need to be changed? Approaches from the Mining Software Repositories (MSR) community help support this question by using the version history of a software system. An approach in MSR is to analyze commits, i.e., co-changed artifacts checked in together and associated metadata (e.g., date and text message), stored in software repositories to infer evolutionary couplings between artifacts [Kagdi, Collard, Maletic 2007a; Zimmermann et al. 2005]. Evolutionary couplings are then used to infer prediction rules of the form \( e_1 \Rightarrow e_2 \). Such rules indicate that should the software entity \( e_1 \) change, the entity \( e_2 \) is also likely to co-change. The level of syntactical representation of source code entities and their changes in a prediction rule is termed as its expressiveness. The effectiveness of a prediction rule
is described in terms of accuracy (i.e., precision and recall) of the estimated changes as well as the computational cost.

An observation from a survey of MSR approaches [Kagdi, Collard, Maletic 2007a], conducted during the prologue of this work, is that the expressiveness of evolutionary couplings among source code artifacts, and hence prediction rules, is at a fairly high-level physical (e.g., file and line) or logical (e.g., class and method) granularity. The precision and recall results of the rules are also considerably low [Ying et al. 2004; Zimmermann et al. 2005]. To improve on the expressiveness and effectiveness, the thesis describes a software-change prediction approach that automatically mines evolutionary couplings between source code entities at the fine-grained syntactic levels. Here, fine granularity refers to source code entities that are not only restricted to fairly high-level structural units such as classes and methods, but also syntactic constructs such as control statements, preprocessor directives, and even comments. Change prediction rules are then formed from these evolutionary couplings. Furthermore, the use of source code dependencies (e.g., in program dependency and call graph) from the Software-Change Impact Analysis (IA) to refine and/or augment the prediction rules is also investigated.

The research conjecture is that software-change prediction using the proposed model results in an overall improved expressiveness and effectiveness. The examination of this conjecture, partially, is substantiated with the research question: How much does the fine-grained expressiveness add to the effectiveness of the prediction rules?
4.1 Relevant Research in MSR

A few representative approaches in MSR that are relevant to our work are briefly discussed. Zimmerman et al. [Zimmermann et al. 2005] used CVS logs for detecting evolutionary coupling between source code entities. Association-rules based on itemset mining were formed from the change-sets and used for change-prediction. Yang et al. [Ying et al. 2004] used a similar technique for identifying files that frequently change together. Gall et al. [Gall, Hajek, Jazayeri 1998] used window-based heuristics on CVS logs for uncovering logical couplings and change patterns, and German et al. [German 2004a] for studying characteristics of different types of changes. Hassan et al. [Hassan, Holt 2004] analyzed CVS logs for the purpose of software-change prediction. The proposed approach differs from these works. The differences are highlighted in the discussion in Section 4.2.

Fluri et al. [Fluri, Gall, Pinzger 2005] analyzed change-sets from a CVS repository to distinguish changes within source code entities such as classes and methods (termed as structural changes) from the changes to license updates and white space between source code entities (termed as non-structural changes.) Their study on an Eclipse plugin found over 31% change-sets with no structural changes and over 51% change-sets with at least one non-structural change. In one of the rare cases, Ying et al. [Ying et al. 2004] defined the interestingness measure of the evolutionary coupling based on the source code dependencies such as calls, inheritance, and usage. Their study on Eclipse and Mozilla found evolutionary couplings that were not represented by the source code dependencies they considered. However, a software-change prediction approach
that combines both IA and MSR techniques and their systematic evaluation is not reported in the published literature. For a thorough description of the area of MSR, refer to the Kagdi et al. [Kagdi, Collard, Maletic 2007a] comprehensive survey and taxonomy of a large number of MSR approaches in the context of software evolution.

4.2 The Approach

Broadly, the proposed approach for mining fine-grained evolutionary couplings and prediction rules consists of three steps,

- Extract the change-sets stored in software repositories
- Process the changes in the files of extracted change-sets at various levels of syntactic constructs
- Apply a data mining technique to automatically infer evolutionary coupling and then form change prediction rules

4.2.1 Change-sets in Software Repositories

Source code repositories store metadata such as user-IDs, timestamps, and commit comments in addition to the source code artifacts and their differences across versions. This metadata explains the why, who, and when dimensions of a source code change. Modern source-control systems, such as Subversion, preserve the grouping of several changes in multiple files to a single change-set as performed by a committer. Version-number assignment and metadata are associated at the change-set level (and not to each file that is changed as is in the case with some version-control systems such as CVS) and recorded as a log entry.
A log entry in a Subversion repository corresponds to a single commit operation. This log information can be readily obtained by using the command-line client `svn log` and a number of APIs (e.g., pysvn.) Subversion’s log entries include the dimensions author, date, and paths involved in a change-set. Additionally, a text message describing the change entered by the developer is also recorded. Note that the order in which files appear in a log entry is not necessarily the order in which they were changed. In the rest of the chapter, we use the term change-sets for the log entries and their corresponding files in Subversion repositories.

### 4.2.2 Processing to Fine-grained Change-sets

The differences in a file of a change-set could be readily obtained at a line-level granularity (e.g., `diff` utility.) These line differences in the files need to be mapped to the corresponding fine-grained differences in the syntactic constructs. The proposed approach employs a lightweight methodology for fine-grained differencing of files in a change-set. The previous and current versions of a source code file are processed using a word-differencing tool, namely `dwdiff` (http://os.ghalkes.nl/dwdiff.html.) This differencing produces two source code files along with the changed locations. The first file is marked with the exact locations from where tokens, i.e., words of a programming language, are deleted and the second file is marked with the exact locations where tokens are added. These markers are appropriately labeled with “specialized” source code comments.

Both files produced from the word differencing are converted to the srcML representation [Collard, Kagdi, Maletic 2003]. srcML is an XML representation of source
code that explicitly embeds the syntactic structure inherently present in source code text with XML tags. The format preserves all the original source code contents including comments, white space, and preprocessor directives. Finally, both srcML files are proceed with the standard XML processing tools to give a list of added and deleted constructs in a hierarchical manner up to the granularity of an identifier. The entire process is realized in the form of a fine-grained differencing tool, namely codediff. The tool codediff is used to process all the files in every change-set for source code differences at a fine-grained syntactic level.

The codediff approach has a very close similarity to Collard’s srcDiff representation [Collard 2004] that achieves fine-grain differencing using line differencing (i.e., diff) and srcML. The important distinction between the two is that codediff achieves much finer levels of difference granularity than the srcDiff toolset and avoids situations of a line change cross-cutting multiple constructs. Alternatively, heavyweight approaches such as AST based and semantic comparisons are not practically feasible due to a very high computational cost involved in processing a number of versions [Mens 2002; Raghavan et al. 2004]. Additionally, they typically require a system-wide parsing and as such may need additional files that are outside a given change-set to the extent of the entire system.

4.2.3 Mining Fine-grained Evolutionary Couplings

The granularity and composition of a change-set committed to a Subversion repository may vary across tasks, developers, and projects. Therefore, a single high-level change may be completed over multiple change-sets (e.g., a refactoring can be committed
in one or more change-sets.) In order to mine larger or more complete patterns, we need to consider single-cohesive changes that spread over a sequence of change-sets. However, the changes-sets corresponding to such changes are rarely explicit (at least not directly recorded in the repositories or documented.)

The approach uses three heuristics to group change-sets [Kagdi, Maletic, Sharif 2007; Kagdi, Yusuf, Maletic 2006]. Each heuristic takes a set of change-sets and forms groups of “related” change-sets. The heuristic Time Interval places all the change-sets committed in a given time duration in the same group. The heuristic Committer places all the change-sets committed by a specific committer in the same group. The heuristic Committer + Time Interval places all the change-sets committed by a given committer within the same time interval in the same group. Therefore, each group formed by heuristics is actually a sequence of change-sets. Change-sets with a larger revision number occur before those with a smaller revision number in a group.

A sequential-pattern mining tool, namely sqminer is developed to uncover evolutionary couplings from the set of groups formed by the undertaken grouping heuristic. The basic premise of sqminer is if the same set of source code entities frequently co-change then there is a potential evolutionary coupling between them. Evolutionary couplings are represented as ordered patterns. These ordered patterns are used to generate sequence rules that serve as prediction rules for source code changes. This approach of sqminer has already been applied previously to mine co-changes at the file level [Kagdi, Yusuf, Maletic 2006], uncover/discover traceability links between
source code and other types of artifacts [Kagdi, Maletic, Sharif 2007], and mine evolutionary couplings of localized web documents [Kagdi, Maletic 2007b].

The application of sqminer for mining fine-grained evolutionary coupling is demonstrated with the aid of an example pattern. This pattern is a real case that was mined from the version history of the kword application in KDE and is given below,

\{(file, KWOpenDocumentLoader.cpp, M)/(comment, φ, D)\} → \{(file, KWOpenDocumentLoader.cpp, M)/(include, φ, D)\} → \{(file, KWOpenDocumentLoader.cpp, M)/(function, KWOpenDocumentLoader::loadOasisText, M)/(block, φ, M)/(if, φ, M)/(then, φ, M)/(block, φ, M)/(comment, φ, A)\}

This pattern was found to occur three times between the period of February 2007 and April 2007. This evolutionary coupling indicates that three entities are likely to co-change. All the changes are in the file KWOpenDocumentLoader.cpp in this pattern. In the first change-set, a comment is deleted, followed by a deletion of an included file in the second change-set, and lastly addition of a comment in an if statement of the function named loadOasisText (possibly a method of the class named KWOpenDocumentLoader) in the third change-set.

Each changed source code entity is represented by recursive tuples of the format (type, [name], change.) The attribute type can be any language construct that is identifiable by srcML. The attribute name is optional for the anonymous constructs such as comments. The attribute change can be A (add), D (delete), or M (modification.) We should note that the tool codediff is capable of enumerating the content of the added/deleted entities in detail (e.g., specific include file deleted in our example.)
However, this level of detail is avoided and only the top-most level is considered because the likelihood of the same entity added or deleted multiple times is practically very low. Overall, the approach allows a combination of the generalized and specific entities in the mined evolutionary couplings so that useful information is uncovered at multiple levels.

Now what is the rationale to believe that mining at a finer granularity level would improve effectiveness (i.e., precision and recall)? When mining at a coarse-level of granularity, changes are considered only at the physical level (e.g., file) or fairly high-level logical source code entities (e.g., class and method.) As a result, for example, prediction rule such as \{(file, f_1, M)\} \Rightarrow \{(file, f_1, M)\} \Rightarrow \{(file, f_2, M)\} could result from mining. This rule states that if the file $f_1$ is modified in any way, the file $f_2$ is also likely to be modified in some way afterwards. Due to a lack of specificity in this rule, sometimes its recommendation would stand correct (at best), and some other times not. Therefore, it could negatively affect an overall precision of prediction and effort required for the examination of false candidates.

However, fine-grained evolutionary rule and prediction rule have much more additional (with regards to any way and some way) information to base their decision before suggesting a recommendation. In our example pattern from kword, the prediction rule would recommend that a comment should be added in the function loadOasisText, after a comment is first deleted and then a include directive is deleted from the file KWOOpenDocumentLoader.cpp. Clearly, such careful and conservative rules could provide much better precision by making a more informed decision. Additionally, they provide much detailed change information needed to be performed to the developer. In
fact, with a closer look, it can be shown that mining at a fine-grain level would not even report potentially less accurate prediction rules. Also, at fine-grain levels there are more entities to consider for prediction (e.g., typically more number of functions than number of files.) Therefore, coverage and eventually recall could also improve. Once again, in case of only file-level mining for the kword example, there is no evolutionary coupling of the file KWOpenDocumentLoader.cpp with other files, and thus the couplings among its constituent constructs would go undetected and unsupported for prediction.

The sqminer approach to mining evolutionary coupling differs from the most comparable approach given by Zimmermann et al. [Zimmermann et al. 2005]. Their mining is restricted to a single change-set level (and not a sequence of change-sets.) Also, they processed change-sets to only file, class, method, and variable level granularities (and not control structures, preprocessor directives, comments, etc.) As such, their prediction rules are fairly coarse grained. For example, their approach cannot identify the example pattern from kword.

4.3 Evaluation Strategy

The objective here is to investigate an empirical answer for the question: How much does the fine-grained expressiveness add to the effectiveness of the prediction rules? Open source systems such as KDE (K Desktop Environment), Apache, jEdit, and GCC (GNU Compiler Collection) will be used as subject systems. These systems provide a variety of applications, domains, programming languages, development practices, and sizes.
Two sets of studies, one at a file level and another at various fine-grained levels, will be conducted. The results of these two studies will be compared to answer our question. For both studies, the general evaluation methodology is to first mine a set of commits from a subject system’s repository for uncovering evolutionary dependencies. We call this the training-set. Next, we select a later set of commits (called the evaluation-set) and see how well they are predicted by the evolutionary couplings in the training-set. This process will be repeated for a number of portions of the subject systems’ versions history (i.e., similar to n-fold cross validation approach in data mining.) As stated before, two widely used metrics precision and recall will be used for measuring the effectiveness. A careful assessment of structural, internal, and external validities including the statistical significance of comparison results will be reported.

4.4 Summary

The work presented here is directed at an investigation of the co-relation between the expressiveness and effectiveness of a software-change prediction approach based on MSR. That is, does predicting changes at a finer granularity of source code constructs improve accuracy/cost? The proposed evaluation will provide an empirical basis to help answer the above questions and provide a recommendation catalogue for different classes of changes in a given scenario.
CHAPTER 5

Combining Single-Version and Evolutionary Dependencies for Change Prediction

Two broad groups of methodologies are described in the literature for supporting software changes. The approaches described under the area of Software-Change Impact Analysis (IA) are among the early efforts that support change estimation [Bohner, Arnold 1996]. Mining Software Repositories (MSR) is a growing area of research that has shown the emergence of approaches for supporting change predictions [Gall, Jazayeri, Krajewski 2003; German 2004a; Hassan, Holt 2004; Zimmermann et al. 2005].

Bohner and Arnold surveyed IA methodologies in 1996 [Bohner, Arnold 1996], and a number of approaches based on improved static and dynamic analyses are proposed thereafter (e.g., [Briand, Labiche, Sullivan 2003; Chen, Vaclav 2001; Law, Rothermel 2003; Moonen 2002; Tonella 2003].) An extensive examination of the MSR approaches was recently completed [Kagdi, Collard, Maletic 2006], of which a preliminary survey of six approaches that support software changes is discussed in [Kagdi, Collard, Maletic 2005]. These works indicate that IA and MSR methodologies can undertake orthogonal perspectives towards meeting the common goal of supporting change management.

5.1 IA and MSR

The term Dependency Analysis is used to refer to impact analysis of software artifacts at the same level of abstraction (e.g., source code to source code and design to design) [Bohner, Arnold 1996]. The basic premise of a typical dependency-analysis
approach is to use the relationships between entities (e.g., files and functions) in an abstraction model (e.g., call-graphs, program-dependency graphs, or UML models), and/or dynamic behavior (e.g., run-time profiling data), of a single snapshot of a program (e.g., program version/release.) A relationship between entities in a model is considered as an indicator of a change dependency between them. That is, if an entity is changed, the “related” entities are estimated to change.

The expressiveness and effectiveness is dependent on the underlying abstraction model(s) and their construction methodology. A model construction may require a complete analysis of all the entities in a snapshot. Also, the estimations are seldom refined for future predictions from the actual changes that occur in the past. In summary, dependency analysis largely remains a single-version activity. That is, the underlying models used to compute the various impact sets takes into account only a single snapshot (most typically the current version) of the program. Dynamic analysis is performed on data collected from executing a single version of the program. Also, it also does not consider the various metadata about a change such as who, why, and when a change was made.

Approaches in MSR support change prediction by using changes performed across multiple versions of a software system that are typically stored in software repositories. In addition to storing differences between artifacts (i.e., file and line number), metadata such as who, why, how, and when associated with a change are also found. A set of versions of the software artifacts is analyzed to uncover pertinent
information and trends of software changes that are then used to predict changes in the latter versions.

One approach in MSR is to analyze commits (i.e., a set of changed artifacts checked-in together) and metadata in software repositories to infer evolutionary dependencies or co-changes between artifacts [Zimmermann et al. 2005]. Since software repositories typically provide differences only at file and line number, the expressiveness of the entities involved in a co-change is dependent on the further fine-grain analysis performed (e.g., to achieve a syntactic and/or semantic level of granularity.) A unique advantage of such a MSR approach is that only changes to entities are analyzed compared to complete analyses of all the artifacts. However, on the other end, such an approach may fail to predict “unseen” changes in the past, and may incorrectly predict obsolete changes to entities that do not exist anymore.

A straightforward step is to combine the two somewhat different approaches of dependency analysis and MSR\(^4\) approach. Using both the software dependencies and evolutionary dependencies could help improve the overall change prediction methodology. The actual changes in a software repository can be utilized to assess the quality of the impact sets produced by impact analysis techniques. Additionally, the historical context can be utilized to augment the impact analysis models to improve their change prediction power. Similarly, software entities that are not predicted to change by

\(^4\) MSR is much broader than supporting software changes. Here, for brevity we limit it to approaches for change analysis and prediction.
MSR but are predicted correctly by impact analysis could be used to validate MSR. Therefore, impact analysis and MSR could be used to cross validate, refine, and supplement each other.

5.2 Supporting a Combined Approach

A hybrid software-change methodology consisting of both dependency analysis and MSR approach requires toolset support for constructing various abstraction models and mining co-changes from version history. We developed an infrastructure that provides a common basis for satisfying both the above requirements.

The abstraction models which are the underlying basis of dependency analysis will be constructed from the srcML representation [Collard, Kagdi, Maletic 2003; Maletic, Collard, Marcus 2002]. srcML is an XML representation of source code that explicitly embeds the syntactic structure inherently present in source code text with XML tags. The format preserves all the original source code contents including comments, white space, and preprocessor directives. The capability and features of srcML representation can be used to easily extract facts [Collard, Kagdi, Maletic 2003] with standard XML processing tools, and derive abstraction models such as call-graphs, program-dependency graphs, and UML models from source code.

A frequent-pattern mining tool, namely sqminer is developed to uncover co-changes from commits stored in a source code repository. sqminer was applied to mine co-changes at the file level [Kagdi, Yusuf, Maletic 2006]. For example, sequences of changed-files (i.e., co-changes) such as \{f1\}→\{f2\} and \{f4\}→\{f5\} were mined. The symbol → in the sequence \{f1\}→\{f2\} indicates that changes in \{f1\} happened before
A differencing tool, namely codeDiff is developed based on srcML and a word differencing tool, namely dwdiff. It takes two versions of a source code file and produces differences between them at a syntactic level. codeDiff will be used to process the differences in a source code file of a commit to fine-grained syntactic level. This will help mining co-changes with sqminer at fine-grain syntactic levels.

5.3 Evaluation Framework

The first part of the evaluation is to select the abstraction model for dependency analysis and the mining methodology for MSR. For dependency analysis, the estimated changes will be based on the abstraction models formed by static analysis (e.g., static call graphs and program-dependency graphs) and dynamic analysis (dynamic call graphs and profiling.) The granularity of entities predicted from both approaches will be appropriately matched. For example, if the expressiveness of estimated entities is functions/methods and variables from dependency analysis, then for MSR the co-changed will be mined at the same granularity.

The version histories of open-source systems such as the KDE (websvn.kde.org/trunk/KDE), Apache, jEdit, and GCC will be used as subject systems. These systems provide a variety of applications, domains, programming languages, development practices, and sizes.

The general evaluation methodology is to first mine a set of commits from KDE repository for co-changes. We call this the training-set. Next we select a later set of commits (called the evaluation-set) and see how well they are predicted by dependency analysis, MSR, and their combination. This process will be repeated for a number of
portions of the KDE versions history (i.e., similar to n-fold cross validation approach in data mining.) Two widely used metrics precision and recall will be used for measuring effectiveness. A careful structural, internal, and external validity will be discussed to provide the context of the results.

Let $R_i$ be the set of entities changed in the commit $i$ of the evaluation-set. Let $D_i$ be the set of entities estimated to change in the commit $i$ of the evaluation-set with dependency analysis. Let $M_i$ be the set of entities estimated to change in the commit $i$ of the evaluation-set with co-change rules (note that sqminer forms association/sequence rules for change predication.)

The changed entities in commits do not have the specific ordering information in which they were changed. Therefore, $D_i$ is taken as the transitive closure of all entities involved in the calls and definition-user relationship with the changed entities in a commit. The set $M_i$ is the set of all the entities predicted by all the applicable co-change rules. The precision and recall of dependency analysis and co-change approach on the evaluation-set are defined as follows,

**Definition:** The precision of dependency analysis, $P_D$, is the mean percentage of correctly estimated changed entities over the total estimated entities.

$$P_D = \frac{1}{n} \sum_{i=1}^{n} \frac{|D_i \cap R_i|}{|D_i|} \times 100\%$$

**Definition:** The recall of dependency analysis, $R_D$, is the mean percentage of correctly estimated changed entities over the total correctly changed entities.

$$R_D = \frac{1}{n} \sum_{i=1}^{n} \frac{|D_i \cap R_i|}{|R_i|} \times 100\%$$
Definition: The precision of co-change approach, $P_M$, is the mean percentage of correctly estimated changed entities over the total estimated entities.

\[
P_M = \frac{1}{n} \sum_{i=1}^{n} \left( \frac{|M_i \cap R_i|}{|M_i|} \right) \times 100\%
\]

Definition: The recall of co-change approach, $R_M$, is the mean percentage of correctly estimated changed entities over the total correctly changed entities.

\[
R_M = \frac{1}{n} \sum_{i=1}^{n} \left( \frac{|M_i \cap R_i|}{|R_i|} \right) \times 100\%
\]

With regards to a combined approach, there is an interesting question. Should the union or intersection of the estimations $D_i$ and $M_i$ be taken for the commit $i$? This question may not be much of an issue, if both $D_i$ and $M_i$ predict the same estimation set. In a different situation, taking their union could result in an increased recall, however at the expense of decreased precision (if the union set has a large number of false-positive estimates.) On the other hand, taking only the intersection imposes a stricter constrain that could result in an increased precision, however, at the expense of decreased recall.

A combined approach for change prediction that uses the union of estimations of dependency analysis and estimations of co-change approach is termed as the \textit{Disjunctive Approach}. The precision and recall are:

\[
P_{D\cup M} = \frac{1}{n} \sum_{i=1}^{n} \left( \frac{|(D_i \cup M_i) \cap R_i|}{|D_i \cup M_i|} \right) \times 100\% \quad \text{and}
\]

\[
R_{D\cup M} = \frac{1}{n} \sum_{i=1}^{n} \left( \frac{|(D_i \cup M_i) \cap R_i|}{|R_i|} \right) \times 100\%
\]

A combined approach for change prediction that uses only the intersection of estimations of dependency analysis and estimations of co-change approach is termed as the \textit{Conjunctive Approach}. Precision and recall are:
\[ P_{D\cap M} = \frac{1}{n} \sum_{i=1}^{n} \frac{|(D_i \cap M_i) \cap R_i|}{|D_i \cap M_i|} \times 100\% \]

\[ R_{D\cap M} = \frac{1}{n} \sum_{i=1}^{n} \frac{|(D_i \cap M_i) \cap R_i|}{|R_i|} \times 100\% \]

The precision and recall of the individual approaches will be used as baselines to assess the effectiveness of the disjunctive and conjunctive approaches. Analysis of the results of disjunctive approach could provide insight into what kinds of changes are “better” predicated by which kind of an approach. The exclusive co-change estimation set \(M_i - D_i\) that has a high precision and recall is of special interest. The entities in such sets are the ones that are only correctly predicated by the co-change mining approach. This may bring forth that change history represents one of the few sources of information available for recovering “hidden” dependencies that is manually created and maintained by the actual developers or dependencies that are accidental. The former kinds embody part of the developer’s knowledge and experience, or consisting of domain-specific couplings. Here, such dependencies are termed as pure-evolutionary dependencies.

The exclusive dependency-analysis estimation set, \(D_i - M_i\) that has a high precision and recall represents change dependencies that could only be correctly predicted by the dependency-analysis approach. Therefore, indicating that co-change mining approach alone may be insufficient. Similarly, very the low-accuracy exclusive sets \(M_i - D_i\) and \(D_i - M_i\) may indicate when not to use co-change approach and dependency analysis respectively.

The analysis of exclusive estimation sets could be combined with change metadata (e.g., commit message, bug/issue report, and committer) present in software
repositories, and change classification taxonomies, for building heuristics. An example of such heuristic that may result is that changes needed to fix a particular kind of bug should be estimated by co-change analysis only.

The common estimation set, $M_i \cap D_i$ and equal estimation set, $M_i = D_i$ is predicted by both approaches. In such a case, heuristics could be developed to favor a particular approach based on accuracy and computational cost. This opens up room for developing effective estimation ranking mechanisms.

### 5.4 Summary

Our goal is to examine whether the combined use of IA and MSR approaches results in an improved expressiveness and effectiveness for software-change prediction. That is, what is the additional gain in the effectiveness of the prediction rules with a further integration of source code dependencies from IA?

We feel that the combination of these approaches will result in more accurate results. An empirical investigation is currently being conducted on a number of open-source systems (e.g., KDE and Apache) to evaluate a hybrid approach that combines the two. Also, a number of tools to support mining and analysis on a fine-grained level are being developed for supporting this investigation. In this discussion, the effectiveness of a hybrid approach is of principal focus.

---

5 Here, the focus is on the source code change prediction, however the examination remains of interest for all software artifacts.
The contributions of this investigation are a step towards answering our overarching research question as to what are the exclusive and potentially synergistic benefits of IA and MSR methodologies with regards to change prediction. The proposed evaluation will provide an empirical basis to help answer this question and provide a recommendation system for different classes of changes. We believe that identification of pure-evolutionary dependency is an interesting and important problem to appreciate the true value of MSR approaches.
CHAPTER 6

Mining Evolutionary Dependencies from Web-Localization Repositories

Open source systems have a global user community that spans across geographical and cultural boundaries. In order to better serve and retain such a diverse user base, open source projects are increasingly developed so that they may adapt to the various natural languages and environments, i.e., they can be localized. Products such as Linux, OpenOffice, and KDE provide localization of user interfaces and user documentation, including online help web pages, in several languages and locales (i.e., characters set, date/time, and currency.)

It is not uncommon for large-scale software to have a great number of online help manuals localized in multiple languages. For example, Figure 4 shows the KSpell (a KDE application) online help manuals in US English and German. As the software evolves due to continual changes (e.g., new feature additions or defect corrections) these localized documents also need to be evolved. Unfortunately, software evolution and localized evolution follow disjoint paths. This is in part due to a different set of teams and contributors performing these two different, yet related tasks in isolation. Typically, the software developers rest the responsibility of localization on the translators, and vice versa. Given this situation, a translator or team leader is faced with a common set of questions related to the impact analysis task during evolution of localized documents:

- Given a change in a specific localized document what other localized documents
need to be co-changed?

• Localization in how many languages is affected?

• How much time will it take to carry out these changes?

Answers to these questions help determine if changes in localized documents should be planned for an upcoming software release. This also assists translators to identify potential out-of-date parts that need retranslation and discard obsolete documents. This activity in the localization process is termed a *string freeze*. Fortunately, in a distributed collaborative development environment such as open source development, localized documents are managed in a way similar to source code. Typically, multiple teams contributing to the translation into a set of languages are involved in localization. The localized documents, like source code, are stored in repositories that are managed by version-control systems (e.g., *CVS* and *Subversion*.) We term repositories that store localized online-help documents as *web-localization repositories*. Such a repository stores every change in every document that is checked-in to a repository. We refer to all the changes made to a specific document during its evolution that are stored in a repository as the *document-history*.

Our premise is that the document histories stored in web-localization repositories can be utilized to extract pertinent information and/or uncover relationships or trends

---

6Here, use of the terms repository, document, and pattern without additional context refers to web-localization repository, documents involved in the translation process, and sequential-pattern of documents involved in translation respectively.
about evolutionary characteristics of localized documents. In order to substantiate this claim, we present an approach that is based on sequential-pattern mining [Agrawal, Srikant 1995] to uncover patterns of documents that frequently co-change in a single language or across multiple languages. These patterns are mined from the web-localization repositories. The recovered patterns provide not only a set of documents that are likely to be retranslated or updated in a single version but also in a series of versions. For example, the pattern \{kexi\_basics.po\}→\{kexi\_database.po\} mined from the KDE localization-document repository indicates that the localized-document \textit{kexi\_basics.po} is changed before the localized-document \textit{kexi\_database.po} in two successive versions. This pattern is found in the change history of thirteen translated documents across three different languages. Therefore, it is an indicative of a strong evolutionary dependency among the involved documents. Such patterns could be used to answer the above questions.

In our work [Kagdi, Maletic 2006], we uncovered patterns of documents involved in the translation process including program localization. From this, we formed a conjecture that information embodied in these patterns is useful for supporting localized document evolution. In the work presented here, we provide a rigorous experimental validation of this conjecture in the context of evolving translated online user guides and other web documents. The evaluation will show that the patterns mined from an earlier part of document histories reoccur with similar frequency in a later part of the document histories. As such, our approach directly supports the impact analysis questions stated above. Furthermore, it assists user communities to anticipate updates to the
documentation in their respective languages based on the past translation patterns. Additionally, we believe that the patterns discovered by our approach could encourage/aid localization efforts in a new natural language by providing the translation patterns in other languages.

Figure 4. Online manuals for the KDE application KSpell in languages a.) US English and b.) German.

6.1 Document Localization: the GNU Model

A number of open source projects use the *gnu gettext* model for localization purposes [GNU 2006]. Such projects produce source code (i.e., string literals) and documentation in a base language (typically US English), and then extract the strings and documentation that require a language-specific translation, and finally translate them to another language. Localization is a semi-automatic process in which the translation
process, i.e., converting text in a document from one language to another, largely remains a human-intensive activity. Actually localization is only a part of the Native Language Support (NLS) paradigm. Establishing NLS in a system encompasses Internationalization (better known as i18n) and localization (better known as l10n) [GNU 2006]. Internationalization is a generalization process that gives a program the ability to understand and support multiple locales (e.g., interaction messages, input, output, date/currency formats in multiple languages.) Localization is a specialization process that produces a locale-specific instance from an internationalized program. Though, NLS (specifically localization) requires a non-trivial human effort, tools such as gnu gettext and KBabel help simplify the task.

Multilingual online help documents of open source systems typically follow the same internationalization and localization process that is used for the source code of a project. Typically documents are produced in a base language (typically US English.) A translation team localizes these base documents to another language (e.g., German.) The translation process that follows gnutext model uses the Portable Object (PO) files [GNU 2006] as a basic representation for localization. A PO file consists of textual entries. Each entry is a pair of untranslated and translated strings. An untranslated document (i.e., the parts of it that require internationalization) is converted to a corresponding PO file. Such PO files typically contain an empty translated string in the pair of strings and are stored as POT (Portable Object Template) files. These template files are then used to produce a language-specific translation by manually filling in the empty translated strings. The language-specific copies of these documents are then converted back to the
original format of the documents. With the context of documents in the **gnu gettext** model, we define the following terms,

**Definition**: *Base-documents* are documents written in a language that does not need translation or localization (typically US English.)

**Definition**: *Internationalized-documents* are the documents formed from the parts or whole of the base documents that serve as customization templates for translation to a set of other languages (POT files.)

**Definition**: *Localized-documents* are the documents that are derived from the internationalized documents and translated to a specific language (PO files.)

**Definition**: *Translated-documents* are language-specific localized documents equivalent to the base documents (e.g., German and Hindi.)

**Definition**: *Message-pair* is a pair of a string in base-language and its translated string in another language (e.g., a pair of *msgid* and *msgstr* in PO/POT files.)

We now explain the document localization process by using a simple, albeit concrete, example from the **KDE** project. In the case of **KDE**, the base-documents are written in the US English. The ubiquitous *docbook* representation (www.docbook.org) is used for authoring base-documents, and then used for transforming documentation in various formats including the HTML manuals. A number of standard tools (e.g., *XSLT* and *xsltproc*) exist to facilitate this transformation task. The online manuals shown in Figure 4a and Figure 4b are generated with such a transformation from their respective docbooks. Consider a portion of a base-document shown in Figure 5a. The content of the element *title* is in the language US English. In order for this document to support
native language, it is converted to an internationalized-document (POT format) as shown in Figure 5b. An entry \((msgid, msgstr)\) of strings is created. The string prefixed with \(msgid\) contains the content of the element \(title\) from the base-document and the string prefixed with \(msgstr\) is left empty. This conversion is facilitated by the tool \(xml2po\). The tool also inserts additional context information in the form of comments. For example, the comments (lines starting with the \(\#\)-symbol) provide information about the element name whose content is translated and its location in the original document. An instance of localized-document (i.e., a PO file) is created from the internationalized-document. In this case, the string \(msgid\) is manually translated to German language and entered in the string \(msgstr\) as shown in Figure 5c. Finally, the localized-document is converted to the translated-document \((docbook\) format with textual contents in German) as shown in Figure 5d.

As the software evolves, the base-documents may need to be changed due to the changes in the software. Now, we discuss the evolution of documents with regards to their localization, where the evolution data is stored, and how that data can be acquired and used to support their further evolution.

6.2 Evolution of Translated Documents

The evolution of translated web documents is analogous and closely coupled to source code evolution. The task of performing a single high-level logical change in source code is either planned activity (e.g., addition of a new feature or a refactoring), unplanned activity (e.g., fixing an unforeseen side effect due to a change), or a combination of both. A typical planned change is implemented in small increments with
the goal of maintaining the overall system in a coherent state (e.g., preserve the build or compile-able state, change source code and documentation in separate steps.) However, such is the nature of software that an extremely well planned change may lead to further unanticipated changes. A single logical change can crosscut multiple source code locations, and require a large sequence of changes in the source code for its complete realization. Consequently, such a single high-level source code change may cause a number of changes to multiple base-documents.

Figure 5. Excerpts of a.) the base-document man-kdeoptions.7.docbook from kdelibs in US English and d) its corresponding translated-document in German. Excerpts of b.) internationalized-document for translation to other languages (POT file) and c.) localized-document in German (PO) used in the translation process.

The translation teams work towards keeping the corresponding translated-documents synchronous with the changes in the base-documents. The evolution of documents that are involved in the translation, like source code evolution, is a continuous
process in which changes are performed incrementally. That is, (re)translation is often performed for a group of “related” message-pairs, possibly spanning across multiple documents, in a single step. Once again, a sequence of groups, of related message-pairs, may change over time to accommodate the changes in base-documents. Changes in base-documents result in the following change scenarios:

- **Additions (A):** New internationalized-documents, triggering addition of corresponding new localized-documents and translated-documents.
- **Deletions (D):** Obsolete internationalized-documents, eventually causing deletion of corresponding localized-documents and translated-documents.
- **Modifications (M):** Updates to internationalized-documents that lead to: addition ($M$-$A$) and/or deletion ($M$-$D$), and/or modified ($M$-$M$), message-pairs in corresponding localized-documents.

A change in an internationalized-document (POT file) may ripple changes in multiple localized-documents across a number of languages. While the scenarios $A$, $D$, $M$-$A$, and $M$-$D$ are generally straightforward to handle, the scenario $M$-$M$ often needs a careful consideration. A given $M$-$M$ change could invalidate translations in existing message-pairs, thus requiring them to be retranslated. The use of tools such as *gettext* and *KBabel* in the document evolution is analogous to the use of tools *diff* and *patch* in the context of source code evolution. They provide assistance in merging the changed message-pairs from the internationalized-documents with the corresponding message-pairs in the localized-documents. These tools help identify message-pairs in the localized-documents that should be considered as retranslation candidates ($M$-$M$), add
new message-pairs that should be translated for the first time \((M-A)\), and mark the message-pairs that have possibly become obsolete \((M-D)\).

However, the human translators are the ones who finally decide on how to handle the affected message-pairs, and manually carry out the appropriate changes including (re)translation. There is no single standard that is universally accepted and practiced in managing this task. The process in which the affected message-pairs are changed depends largely on the translator’s discretion. This includes, for example, checking which localized-documents need changes with regards to changes in the internationalized-documents (with tool support), which message-pairs should be changed together as a single related group, how many groups should be formed, and the order in which these groups should be changed to align with the equivalent changes in base-documents. Clearly, this could vary according to the translator’s experience, skills, and overall project state and policies.

Fortunately, the situation is not completely abysmal. Documentation and their localization is a team activity in large open source projects. It is not uncommon to have multiple contributors developing the documentation and translating the same part of the system. Therefore, version-control systems are commonly used to coordinate and manage these efforts. This historical information about the evolution of the translated documents is often captured in web-localization repositories. These web-localization repositories, like source code repositories, are managed by version-control systems such as Subversion (http://subversion.tigris.org/) or CVS (www.nongnu.org/cvs/). Additionally, metadata such as user-ids, timestamps, and commit comments are often
times stored. This metadata can explain some of the why, who, and when characteristics of a performed change. Let us now examine the information found in the repositories and how it can be obtained. We first start with definitions that are relevant to the discussion.

**Definition:** A *change-set* is a set of changed documents that are checked-in together to a repository in a single commit operation.

**Definition:** A *revision-number* is an identifier used by version-control system to track the state of documents at a given point in time (i.e., a version.) Used synonymously with the term *version-number*.

**Definition:** A *logentry* is a record of metadata associated with the change-set or its member document by a version-control system.

In a repository managed by *Subversion*, version-number assignment and metadata are associated at the change-set level and recorded as a logentry. *Subversion*’s logentries include the attributes *committer* (the person who checked-in the change-set), *date* (the change-set checked-in data and time), and *paths* (files and directories in the change-sets.) Each change-set (logentry) is uniquely identified by a revision-number. Figure 13a shows four logentries (i.e., change-sets) of a hypothetical web-localization repository. Revision 1 consists of change-set \{a, b, d\} with three documents a, b, d committed together. Besides metadata, *Subversion* provides access to any version of the document and the difference between any two given versions of a document.

We should note that modern source-control systems, such as *Subversion*, have several improvements over systems such as *CVS* as they preserve the grouping of several changes in multiple files to a single change-set as performed by a committer (i.e., an
atomic commit.) Preservation of a change-set as an atomic commit in repository gives the ability to iterate through the change history at the change-set level (i.e., “undo” at the change-set level rather than the individual file level.) This encourages the practice of committing a set of related changes in a single logical change. This is a standard Subversion policy of the KDE project.

The logentries can be readily obtained from the repository via standard version-control system commands. We term this as change-set-oriented view of the change history. However, as we discussed earlier, the granularity and composition of a change-set may vary across tasks, developers, and projects. Therefore, a single high-level change may be completed over multiple change-sets. The number of documents (i.e., size) may vary across the change-sets throughout versions history. On one end, some change-sets may contain only a few documents that are changed slightly (e.g., only a single message-pair is translated in a single document.) This is a case when a single logical change is performed incrementally and is completed by committing multiple versions. On the other end of the spectrum, you can have a change-set that contains a large number of documents that are completely translated. This is a case where the entire task is completed and then all the changed documents are committed in a single version. Also, the order in which these documents appear in a logentry is not necessarily the order in which they were changed. Simply considering a single logentry is insufficient to determine all the related documents that are typically changed together and the specific (temporal) ordering of the documents involved in a change-set.
In order to uncover a complete change performed for a single high-level change, we need to consider changes that spread over a sequence of change-sets. However, the change-sets corresponding to such changes are rarely explicit, at least not directly recorded in the web-localization repositories (or any type of repository generally), or clearly documented. Note that the change-sets stored as atomic commits in web-localization repositories are serialized. The order in which log entries appear in the log files is controlled by a version-control system. Two unrelated change-sets committed approximately at the same time may appear next to each other. Therefore, treating successive change-sets in the web-localization repositories as related to a single high-level change may prove to be meaningless. Next, we describe our approach to uncover evolutionary dependencies from the change-sets in web-localization repositories.

Figure 6. a.) An example of four revisions available directly from a web-localization repository and b) the change-set-oriented history converted to a document-history.
6.3 Automatically Uncovering Evolutionary Couplings

Broadly, our investigation is about how we can approximate related changes in documents that represent a single cohesive high-level change along with the ordering information. The information that can be utilized is the (serialized) change-sets committed in a specific temporal order and the metadata in the repository. In an effort to obtain complete coverage of the documents that typically change together over multiple change-sets, we take an alternative view of history compared to what is directly available from the version-control system. Furthermore, on this view we employ sequential-pattern mining to uncover evolutionary couplings.

6.4 Document-Change History

The history of documents available from repository is in a change-set-oriented view. However, as discussed in Section 6.2, change-set-oriented view may not provide the complete coverage of documents that are changed to realize a single logical change and may produce meaningless patterns. Therefore, we take an alternative approach of rearranging the change-set-oriented view of history into a view that gives the history of a document with regards to all the change-sets in which it is involved. All these change-sets are related as they contain changes to the same document. We term this alternate arrangement of history as document-history.

In this view of document-history, a document contains a sequence of all change-sets in which it is involved. The ordering in a sequence is based on the revision numbers. A change-set in a sequence with a lower revision number is assigned a position before a change-set with a higher revision number. For the example shown in Figure 13a, the
corresponding document-history is shown in Figure 13b. The history of document \( a \) consists of a sequence of change-sets from revisions 1, 2, 3, and 4. Notice that a change-set in a document-history may contain additional documents that are also co-changed. We now ask the question: Are the co-changes in a single document-history sufficient to infer that such co-changed documents have an evolutionary dependency? Our hypothesis is that if the same co-changed documents occur in a number of document histories (i.e., frequently) then it is likely an evolutionary dependency.

### 6.5 Uncovering Evolutionary Dependencies

Our approach automatically uncovers sets of documents that are typically co-changed in the translation process from the document histories. These uncovered evolutionary couplings can be used to support future changes in localized documents by helping to answer the question: Which other documents need to be changed due to a change in a given document? A frequent-pattern mining technique from the data mining community is used in the approach. This technique does not require any predefined rules to infer co-changes. In fact, coming up with predefined rules may not be always practical due to the latent nature of change practices among developers. These practices are rarely documented and enforced, and as such predefined rules may not be kept up-to-date with the system evolution. Moreover, our approach is not restricted to a predefined set of specific documents that are included/excluded for mining. Alternatively, our approach examines the documents in web-localization repositories and automatically infers evolutionary couplings of documents involved in the translation process of a software
system. Now, we describe the specific frequent-pattern mining technique used in our approach.

The sequential-pattern mining technique from the data mining community can be adopted to accomplish the task of discovering frequently co-changed documents. The general problem of sequential-pattern mining from any dataset takes a given set of sequences (composed of items) and finds all the frequently occurring subsequences (i.e., ordered patterns) that have at least a user-specified minimum support [Masseglia, Teisseire, Poncelet 2005]. Sequential-pattern mining techniques are typically applied to datasets with temporal or other ordering information. For example, in case of market-basket analysis with the additional timestamp information, sequential patterns such as customers who bought a camera are also likely to buy additional memory in the next month. Before we describe the sequential-pattern mining approach in the context of our problem, data-mining terminology that is relevant to the discussion is introduced. The input data to frequent-pattern mining algorithms are in the form of transactions (e.g., customer baskets or items checked-out together in market-basket analysis.) The number of transactions in which a pattern occurs is known as its support. The basic idea is that if the support of a pattern is at least a user-specified minimum support then it is a frequent pattern in the considered dataset.

Formally, the problem of finding frequent sets of sequences is defined as given a set of items, $\alpha = \{i_1, i_2, \ldots, i_m\}$, and a set of transactions, $\tau = \{T_1, T_2, \ldots, T_n\}$, find all the sets of sequences, $S = \{S_1, S_2, \ldots, S_o\}$, that co-occur in at least a given number (or percentage) of transactions i.e., it satisfies a given minimum support, $\sigma_{min}$. Each
Transaction contains an ordered list of events and is identified by an unique id, $T_i = (tid, e)$ where $e = [E_1, E_2, \ldots, E_p]$ $\forall_{i,j} E_i \rightarrow E_j$ and $\rightarrow$ is a given ordering relation on events.

Each event contains a set of items and is identified by a unique id, $E_i = (eid, \subseteq \alpha)$. Each sequence is defined as an ordered list of elements (i.e., itemsets), $S_i = [I_1 \rightarrow I_2 \rightarrow \ldots \rightarrow I_p]$ $\forall I_i \subseteq \alpha$, and each member of an element, $i_j \in I_i$ is defined as an item of a sequence. A mined sequence $S_i$ is called a frequent sequence.

We pose the problem of mining patterns of documents that frequently co-change, i.e., evolutionary dependencies, as an instance of the general problem of sequential-pattern mining. To derive this instance from the general problem, we need to map the general concepts such as item, event, and transaction to their counterparts in our problem’s context. Here, an item corresponds to a document that is found in at least one change-set in a considered (part of the) version history, an event corresponds to a single change-set, and a transaction corresponds to a sequence of change-sets of a single document. The ordering of a sequence of change-sets in a transaction is defined by the ordering of revision-numbers. The documents in the same change-set are unordered, whereas, the documents in different change-sets are ordered according to their revision-numbers. For example, the documents $d1$ and $d2$ in the same change-set with the revision-number 1 occurs before the documents $d3$ and $d4$ in another change-set with the revision-number 2. However, documents $d1$ and $d2$, and $d3$ and $d4$ that occur in the same change-set are left unordered.

Application of sequential-pattern mining to our problem automatically produces a set of frequent sequences. That is, a frequent sequence occurs in at least a user specified
number of documents histories. Here, an evolutionary dependency between documents is represented as a frequent sequence. We refer to these frequent sequences as ordered patterns. An ordered pattern is made up of ordered elements. Each element is made up of unordered items. The ordering of elements imposes a partial order on the items. For example, the pattern \{d1, d2\}→\{d3, d4\}→\{d5\} is made up of 3 elements and 5 items. It indicates that the element \{d1, d2\} happens before the element \{d3, d4\} and the element \{d3, d4\} happens before the element \{d5\}. However, the happens-before relation between items d3 and d4 is unknown in the element \{d3, d4\}. Therefore, an ordered pattern can (indirectly) establish both the ordered and unordered relationship between items. In context of our problem, an element of a pattern maps to a change-set or its subset. Therefore, an element of a pattern is the set of documents that change in the same reversion or change-set. Elements are ordered according to their version-numbers. An element with a higher version-number occurs after another element with a lower version-number.

We have developed a sequential pattern-mining tool [Kagdi, Yusuf, Maletic 2006], namely sqminer, that is based on the Sequential Pattern Discovery Algorithm (SPADE) [Zaki 2001] which utilizes an efficient enumeration of ordered patterns based on common-prefix subsequences and division of search space using equivalence classes. Additionally, it utilizes a vertical input-transaction format (i.e., a set of transactions for each item vs. a set of transactions consisting of items) for efficient counting of support values. To help prune the number of candidate patterns produced by the mining techniques, patterns with redundant information are eliminated. A pattern that is frequent
means that all possible patterns formed from the subsets of its items are also frequent.
The support of a pattern is always less than or equal to the sub-patterns. A common
pruning mechanism used in frequent-pattern mining is to eliminate all the sub-patterns
that have the same support of the corresponding (larger) pattern. Such sub-patterns are
only used with other larger patterns and not independently. Therefore, they give
redundant information that may be of a very little meaning. As a result, only disjoint
patterns (i.e., patterns with no common calls) that subsumes all the subsets patterns with
the same support, and subsets of patterns that have higher support values are retained.
Such patterns are known as closed patterns. Our approach produces only closed patterns.
Frequent-pattern mining algorithms typically report the support of a pattern but not the
transactions in which it occurs. The transaction(s) in which a pattern is found is also
recorded.

<table>
<thead>
<tr>
<th>path/a</th>
<th>1 a b d</th>
<th>path/a</th>
<th>2 a b c f</th>
</tr>
</thead>
<tbody>
<tr>
<td>path/c</td>
<td>2 a b c f</td>
<td>path/c</td>
<td>3 a b c e</td>
</tr>
<tr>
<td>path/a</td>
<td>3 a b c e</td>
<td>path/c</td>
<td>4 a c b</td>
</tr>
<tr>
<td>path/a</td>
<td>4 a c b</td>
<td>path/d</td>
<td>1 a b d</td>
</tr>
<tr>
<td>path/e</td>
<td>3 a b c e</td>
<td>path/f</td>
<td>2 a b c f</td>
</tr>
</tbody>
</table>

**Figure 7.** Transactions in sqminer’s input format. Each event is specified on a
separate line and consists of a document name along with its path, version-number,
and the set of documents co-changed.
6.6 Example

Now we demonstrate the automatic mining of patterns with *sqminer* on the document-history shown in Figure 13b. This document-history is first converted to the input-transaction format of *sqminer*. In this format, transactions are specified in the form of events. Each event is specified as a three-unit tuple on a separate line. The first unit is the document name along with the complete path, the second unit is the version-number of a change-set, and the third unit is the set of all documents in a change-set. Figure 7 shows the transactions of the document-history of Figure 13b. For example, the first event is for the document *path/a* that occurs in the revision 1 consisting of a change-set with the documents *a, b, and d*. Here, multiple events are specified on the same line for the sake of brevity. Our approach automatically determines that six documents *a, b, c, d, e, f* are changed in this example of version history and therefore forms six corresponding transactions or document-histories.

The transactions of Figure 7 are then input to *sqminer*. We will use a minimum support of two for a candidate pattern, i.e., at least two document-histories must contain the pattern. Table 12 shows a total of 11 uncovered patterns. The columns *Support*, *Versions*, and *Documents* give the number of document-histories in which a pattern occurs, the number of versions/change-sets committed to complete a pattern, and the number of documents involved in a pattern respectively. Five patterns occur in two document-histories, another two in both three and four document-histories, and one each in five and six document histories. Five patterns take a single change-set or version to complete (i.e., unordered), whereas, others take more than one version (ordered.) Also,
only one pattern consists of two documents, whereas, others consist of more than two
documents. Notice that sub-patterns such as \{a\}, \{b\}, \{a\}→\{c\} and \{c\}→\{a, b\} are not
reported individually as they are closed by larger patterns with the same support values.

**Table 12. Patterns produced by sqminer from the transactions of Figure 7.**

<table>
<thead>
<tr>
<th>No.</th>
<th>Pattern</th>
<th>Support</th>
<th>Versions</th>
<th>Documents</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{a}→{a, c, b, f}→{a, c, b, e}→{a, c, b}</td>
<td>2</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>{a, b, d}→{a, c, b, f}→{a, c, b}→{a, c, b}</td>
<td>2</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>{b}→{a, c, b, f}→{a, c, b, e}→{a, c, b}</td>
<td>2</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>{a, b, d}→{a, c, b, e}→{a, c, b}</td>
<td>2</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>{d}→{a, c, b, f}→{a, c, b, e}→{a, c, b}</td>
<td>2</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>{a, b, d}</td>
<td>3</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>{a, c, b, f}→{a, c, b, e}→{a, c, b}</td>
<td>3</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>8</td>
<td>{a, c, b, f}</td>
<td>4</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>9</td>
<td>{a, c, b, e}</td>
<td>4</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>10</td>
<td>{a, c, b}</td>
<td>5</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>11</td>
<td>{a, b}</td>
<td>6</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

While the patterns with the same number of documents and support value are
quite obvious findings, the patterns with number of documents less or greater than their
support values are of greater interest. For examples, the first pattern \{a\}→\{a, c, b, f\}→\{a, c, b, e\}→\{a, c, b\} consists of 5 documents and occurs in two document-histories,
and the last pattern \{a, b\} consists of two documents and occurs in six document
histories. Such patterns are cases where there are documents in a pattern that tend to
change more in relationship with other documents than independently by themselves.
The first pattern consists of five documents \(a, b, c, e,\) and \(f,\) but is supported by only the
version histories of documents \(a\) and \(b,\) and not the version histories of documents \(c, e,\)
and \(f.\) This potentially indicates that the changes to the documents \(c, e,\) and \(f\) are
potentially dependent on the changes to the documents \(a\) and \(b.\) The last pattern consists
of two documents $a$ and $b$, but is supported by the version histories of all the six documents. This potentially indicates the changes to documents $a$ and $b$ are ubiquitous and perhaps finding such patterns add not much value.

6.7 Evaluation

The assessment of our approach is two fold. First, we show that our approach is able to uncover evolutionary-patterns of documents that are involved in the translation process and show the characteristics of these patterns. Second, we show that the mined evolutionary patterns can be used to support further evolution (i.e., supporting impact analysis and change prediction) of these documents with a high accuracy.

![Figure 8. The structure of the KDE localization repository.](image)
6.8 Document Repository and Tool Set

The open source KDE (K Desktop Environment) system is used for evaluation. The KDE system is actually a collection of applications that represent a wide spectrum of domains, programming languages, sizes, and developers. The documents involved in the translation process are stored in the localization repository, called l10n, the structure and organization of which is shown in Figure 8. A top-level directory is assigned for each language (e.g., the directory fr for the language French) and the internationalized-documents (i.e., the directory templates.) The directories messages, docmessages, and docs are used for the program localized-documents, user guide localized-documents, and the translated-documents respectively. The internalized-documents for program and user guide localizations are stored in the directories messages and docmessages respectively. Both the internationalized-documents and localized-documents are organized into a separate directory for each package, and a file represents a specific application within a package (e.g., kchart.pot, kchart.po.) The translated-documents for each application are stored in a directory typically with the same name as the localized-document file (e.g., the directory kchart.) The directory webmessages contain documents used in the translation of the KDE website (www.kde.org.)

In this repository over 50 different languages are managed for translation. Online user guides in more than 20 languages are generated for a number of applications. We considered a subset of the l10n repository (from the trunk) between the time periods of 2006-01-01 and 2006-03-03 for mining patterns. There are over 4,000 change-sets (i.e., revisions) committed during this two-month period. This period represents a typical two-
month period in KDE development. We picked this time period because it was relatively recent but old enough to allow verification of our findings on more recent versions of the system. The change-sets in this period consist of changes to over 21,000 different documents. These documents are changed over 80,000 times. This indicates that on an average a document is changed approximately three times and an average change-set consists of about five documents. Only change-sets that consisted of less than or equal to ten documents are considered. This is done in order to discard noisy change-sets such as those updating the license information and/or merging/copying. This is commonly used pruning method for mining software repositories [German 2004a; Hassan, Holt 2004; Zimmermann et al. 2004; Zimmermann et al. 2005].

The KDE document repository is managed by Subversion. We developed a change-set extraction tool, namely changeextractor that uses pysvn (i.e., Subversion API for Python) to extract change-sets from a Subversion repository without using a working copy. The tool changeextractor takes a repository URL, start date, and end date of a history, and extracts the logentries from the repository for a specified period between the start and end dates. Furthermore, the tool changeextractor rearranges the logentries in the document-history view of a repository.

With the logentries represented in the document-history view, our sequential-pattern mining technique is applied to find all the frequently occurring patterns in the document-histories. The document-history view of a repository constructed by the changeextractor is fed to the mining tool, sqminer. Mining frequent ordered patterns with sqminer produces a set of closed frequent (ordered) patterns. Additionally, a
pattern’s support and documents histories in which it is found are reported. The configuration parameters of *sqminer* include support, maximum pattern size (i.e., number of items in a pattern), and output of patterns in a XML format. Further detail on the XML format of the ordered patterns can be found in Kagdi et al. [Kagdi, Yusuf, Maletic 2006].

6.9 Mined Patterns and Their Characteristics

![Distribution of patterns with regards to the number of versions taken to complete changes in them and frequency distribution of pattern sizes.]

Since online web documents are of focus here, we considered internationalized-documents, localized-documents, and translated-documents in the directories *docmessages, docs*, and *webmessages* for mining patterns. In our previous work [Kagdi, Maletic 2006] on this topic we also included program-localization documents (e.g., *GUI messages*) in the directory *messages*. We found that program-localization documents exhibit much more frequent change activities than other localized documents. Mining for highly frequent patterns considering both types of localized documents typically results
in a very few patterns that are exclusive to user documentation. Therefore, we excluded program-localization documents from the analysis presented here to achieve focused results and analysis of the user-document localization.

We used sqminer to perform mining with a minimum support of two. That is, a pattern must occur in at least two document-histories, not necessarily in the same language, to be considered as frequent, and thus an evolutionary coupling. We uncovered 841 patterns from the considered KDE l10n repository. Figure 9a and Figure 9b show the frequency distributions of patterns in terms of versions and sizes. The number of versions (i.e., elements) of a pattern is the total number of versions that were committed to complete all the changes to all its constituent documents (i.e., size.) This information about an evolutionary dependency can serve as an indicator as to how many documents are likely to co-change and the estimated number of versions taken to complete the changes in all the co-change documents. Additionally, this information is important to the decision making process with regards to allowing changes during a string (hard) freeze. That is, how many documents (size) are likely to be involved in a change and how much time (versions) it will take to complete changes in them.

Figure 9a shows that about 37% (314), 42% (356), and 20% (167) of the (841) mined patterns were completed in one, two, and three versions respectively. We also found four patterns that took four versions to complete. As can be seen almost 63% of the mined patterns take more than a version. This supports the existence of evolutionary dependencies that proliferate for longer than a single version. Approaches that consider only a single change-set and are used for software-change prediction, such as
[Zimmermann et al. 2004; Zimmermann et al. 2005], do not account for these types of evolutionary dependencies. Figure 9b shows that the patterns consisting of a single document form about 11% (94) and patterns involving two documents form about 22% (185) of the (841) mined patterns. The rest 67% (562) of the patterns consists of more than two documents. Patterns with as many as 16 documents were uncovered. Clearly, this shows that the evolutionary couplings typically consist of more than two documents. In other words, two or more documents frequently co-change. For example, the pattern 

\{krita.po\}→\{krita_tutorial-starting.po, krita_tutorial.po, kchart.po\}→\{krita_introduction.po, krita_tutorial-quick-starts.po, kformula.po, kile.po\}

consists of eight localized-documents and takes three versions to implement all changes.

The change-set of localized-document \{krita.po\} is changed in the first version, followed by the change-set of localized-documents \{krita_tutorial-starting.po, krita_tutorial.po, kchart.po\} in the second version, and finally the change-set of localized document \{krita_introduction.po, krita_tutorial-quick-starts.po, kformula.po, kile.po\} in the third version. These documents are used to generate translated documents of applications in the KOffice package of KDE. Notice this evolutionary dependency crosscuts multiple applications (e.g., krita a graphics editor and kchart a chart and diagram tool.)

The patterns that are common (i.e., supported) in the versions histories of a number of different documents are also of interest. This type of information may support analysis of the amount of retranslation effort required due to potential changes in a common pattern between these document-histories. For example, the pattern \{amarok_config.po, amarok_quick.po\} is common in over 30 document-histories. That
is, changes in the localized-documents *amarok_config.po* and *amarok_quick.po* may potentially cause ripple changes in all these documents. The pattern `{kplato_commands.po} → {kplato_options.po, kplato_wbs.po}` is another example that is supported in 20 document-histories and spans two versions. This type of ripple effect may lead to two versions of each of the 20 documents in which this pattern occurred.

Figure 10 shows the frequency distribution of document-histories with regards to the number of patterns they support. It can be seen that there is a wide distribution of the number of documents supporting the number of patterns and the majority of the patterns have a unique number of supporting documents history. However, there are cases in which more than one pattern is supported by the histories of same number of documents but not necessarily the same set of documents.

Another interesting aspect is to examine different languages that share a common pattern (i.e., the same set of documents are changed in histories of different languages.) This may help perform impact analysis with regards to the number of languages that may be affected due to a change in a common pattern that couples them. For examples, the pattern `{kplato_commands.po} → {kplato_options.po, kplato_wbs.po}` is exhibited in the document-histories in four languages and the pattern `{krita_using-layers.po, krita_using-colorspaces.po}` is exhibited in the document-histories of eight languages. Figure 11a shows the frequency distribution of the 841 uncovered patterns with regards to different languages. Approximately 47% (396) of the patterns are confined to a single language and the remaining 53% (445) of the patterns span two or more languages. There are about 37% (310) of the patterns that are common between two languages but not
necessarily the same pattern and same languages in every case. We found one pattern that is common in ten languages.

Figure 10. The number of common patterns that are found in the number of documents-histories.

Figure 11b shows the number of patterns found in the history of the individual languages. Notice that the same pattern may belong to multiple languages. A large number of patterns are found in the languages such as Italian, Dutch, Spanish, Portuguese, Russian, German, Danish, Swedish, and French. This shows frequent change activities in these languages. Therefore, translated-documents in these languages are likely to be abreast to the base-documents. Languages such as Estonian, Polish, British English, and Brazilian Portuguese begin to show some change activity. From this
data we can surmise that user communities can anticipate translated user guides in all of these languages in the near future.

Figure 11. a.) The number of common patterns found in the histories of different languages, b.) total number of supported patterns by the translation history of a specific language.

6.10 Validating Support for Evolution

We have shown that our approach is able to mine patterns with different characteristics such size, frequency, and occurrences within a single natural language or across multiple languages from a document repository. We now evaluate these patterns by examining their usefulness in supporting changes that occur later. More specifically, we validate these patterns for the task of change prediction. Our general evaluation methodology is to first mine a portion of the version history for patterns. We call this the training-set. Next we mine a later part of the version history called the evaluation-set and see if the results generated from the training-set can accurately predict changes that occur in the evaluation-set. The training-set is the 841 patterns uncovered from the document-
histories in the KDE l10n repository between 2006-01-01 and 2006-03-04. We mined the same repository but for the period between 2006-03-04 and 2006-04-04 using the same mining configuration as in training set. This resulted in 47 patterns being mined from our evaluation set.

The usefulness of the uncovered patterns can be seen in “how well” the training-set predicts the existence of a pattern in the evaluation-set. We validate this using the metrics coverage, recall, and precision.

Let \( T = \{ts_1, ts_2, \ldots, ts_m\} \) and \( E = \{es_1, es_2, \ldots, es_n\} \) be the training-set with \( m \) patterns and evaluation-set with \( n \) patterns respectively. Each member \( ts_j \) where \( 1 \leq j \leq m \) is a pattern in the training-set and each member \( es_i \) where \( 1 \leq i \leq n \) is a pattern in the evaluation-set. Consider the pattern \( es_i = \{f_1\} \rightarrow \{f_2\} \rightarrow \ldots \rightarrow \{f_k\} \) in the evaluation-set consisting of documents \( f_1, f_2, \ldots, \) and \( f_k \) that is undergoing changes. To eventually predict this pattern, the training-set can be queried for candidates after each change-set \( \{f_i\} \) of the pattern \( es_i \) is, or planned to be, performed. Let \( Ces_i = Ces_{i1} \cup Ces_{i2} \ldots \cup Ces_{ik} \) where \( Ces_{ij} = \{ts_1, ts_2, \ldots, ts_p\} \) be a set of candidate patterns suggested from the training-set after the \( j^{th} \) change-set (and all the previous change-sets) of the pattern \( es_i \).

**Definition:** Covered pattern is a traceability-pattern in the evaluation-set for which there is at least one candidate pattern suggested from the training-set,

\[
\text{Covered Patterns} = |\{ \forall es_i \in E \Rightarrow |Ces_i| > 0 \}|
\]

**Definition:** Coverage is the percentage of the total number of covered patterns to the total number of patterns in the evaluation-set,

\[
\text{Coverage} = \frac{\text{Covered Patterns}}{|E|} \times 100\%
\]
**Definition:** Correctly covered pattern is a covered pattern with at least one suggested candidate pattern from the training-set that is the same (completely identical) or its sub-pattern (partially identical.)

**Definition:** Recall is the percentage of the total number of correctly covered patterns to the total number of patterns in the evaluation-set.

\[
Recall = \frac{\text{CorrectlyCoveredPatterns}}{|E|} \times 100\%
\]

Coverage and recall are indicative of the completeness of the training-set in predicting the evaluation-set. Coverage describes how many traceability couplings, a developer can expect to be recommended from the patterns mined in the training-set. Recall describes how many of these recommendations are “correct”. Ideally, both coverage and recall should be 100% (all patterns in the evaluation-set are correctly predicted in training-set.)

Coverage and recall give only one measure of usefulness of the traceability couplings for software-change prediction. An arguably more important measure is the total candidate patterns, both correct and incorrect, suggested from the training-set that require examination for a covered pattern in the evaluation-set.

**Definition:** Relevant patterns of a covered pattern are the number of correctly covered patterns suggested after a change in its given element.

For example let \( es_i = \{f_i\} \rightarrow \{f_2\} \rightarrow \{f_3\} \) be a covered pattern. If the candidate patterns suggested from the training-set are \( \{f_i\} \rightarrow \{f_2\} \rightarrow \{f_3\}, \{f_i\} \rightarrow \{f_2\} \rightarrow \{f_4\}, \) and \( \{f_i\} \rightarrow \{f_5\} \rightarrow \{f_6\} \) after a change in file \( f_i \) in \( es_i \), the relevant patterns are two (out of the
After a change in the file \( \{f_3\} \) in \( es_i \) (i.e., \( \{f_1\} \rightarrow \{f_3\} \)) the suggested candidate patterns are \( \{f_1\} \rightarrow \{f_2\} \rightarrow \{f_3\} \) and \( \{f_1\} \rightarrow \{f_2\} \rightarrow \{f_4\} \). The relevant pattern is one (out of two.)

**Definition:** The *relevance ratio* of a covered pattern is the sum of the ratios of the number of relevant patterns over the number of suggested candidates of all its elements. The relevance ratio in our example is \( \frac{2}{3} + \frac{1}{2} + 1 = 2.167 \).

\[
\text{Relevance Ratio} (es_i) = \sum \frac{\text{relevantPatterns}}{|Ces_i|}
\]

**Definition:** Precision of a covered pattern is the percentage of relevance ratio weighted over its number of elements. Let \( |es_i| \) be the number of elements in \( es_i \).

\[
\text{Precision} (es_i) = \frac{\text{relevanceRatio}}{|es_i|} \times 100\%
\]

Precision of our example is \( \frac{2.167}{3} \times 100 = 72\% \). In the best case, for any given covered pattern in the evaluation-set, only that pattern is suggested from the training-set after changes to any of its elements (i.e., precision is 100%).

Using these metrics, we can evaluate the “goodness” of our approach on the evaluation-set of our KDE study. Notice that precision is measured per pattern. Therefore we give its minimum, maximum, and average values. Coverage and recall are measured for the entire evaluation-set. The coverage and recall of the patterns in the evaluation-set with regards to the patterns in the training-set were found to be 34% and 25% respectively. Only the top ten candidate patterns were suggested for each prediction. The minimum, maximum, and average precision of the patterns were found to be 100%. This result shows that our approach has a high precision in predicting frequent changes. Also restricting candidates to the top ten candidates help prunes false
positives. Notice that coverage and recall has to be the same for precision to be 100% otherwise. Coverage and recall can be considered as low. This could be partially attributed to the general limitation of a purely history-based model that might fail to predict changes unseen in the past. However, a high precision of our approach shows that if candidates are suggested, they are less likely to be false positives. Therefore, we believe that our approach will directly benefit translators, including novice contributors, in their localization efforts.

**Top 10 Translations Teams – trunk**

<table>
<thead>
<tr>
<th>Position</th>
<th>Team Name</th>
<th>Translated</th>
<th>%</th>
<th>Fuzzy</th>
<th>%</th>
<th>Untranslated</th>
<th>%</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Portuguese</td>
<td>79107</td>
<td>100.00</td>
<td>0.00</td>
<td>0.00</td>
<td>79107</td>
<td>0.00</td>
<td>79107</td>
</tr>
<tr>
<td>2</td>
<td>Swedish</td>
<td>78036</td>
<td>98.65</td>
<td>68.09</td>
<td>0.00</td>
<td>1003</td>
<td>1.27</td>
<td>79107</td>
</tr>
<tr>
<td>3</td>
<td>Danish</td>
<td>77912</td>
<td>98.49</td>
<td>0.00</td>
<td>0.00</td>
<td>1195</td>
<td>1.51</td>
<td>79107</td>
</tr>
<tr>
<td>4</td>
<td>Spanish</td>
<td>76285</td>
<td>96.12</td>
<td>450.57</td>
<td>0.00</td>
<td>2632</td>
<td>3.42</td>
<td>79367</td>
</tr>
<tr>
<td>5</td>
<td>Estonian</td>
<td>54865</td>
<td>68.32</td>
<td>7162</td>
<td>8.95</td>
<td>18184</td>
<td>22.73</td>
<td>80011</td>
</tr>
<tr>
<td>6</td>
<td>Italian</td>
<td>53190</td>
<td>67.24</td>
<td>5363</td>
<td>6.78</td>
<td>20554</td>
<td>25.98</td>
<td>79107</td>
</tr>
<tr>
<td>7</td>
<td>German</td>
<td>52720</td>
<td>66.28</td>
<td>5246</td>
<td>6.60</td>
<td>21575</td>
<td>27.12</td>
<td>79541</td>
</tr>
<tr>
<td>8</td>
<td>Russian</td>
<td>50528</td>
<td>63.61</td>
<td>5512</td>
<td>6.94</td>
<td>23392</td>
<td>29.45</td>
<td>79432</td>
</tr>
<tr>
<td>9</td>
<td>Brazilian Portuguese</td>
<td>50352</td>
<td>62.85</td>
<td>6917</td>
<td>8.63</td>
<td>22849</td>
<td>28.52</td>
<td>80118</td>
</tr>
<tr>
<td>10</td>
<td>French</td>
<td>50011</td>
<td>62.76</td>
<td>6295</td>
<td>7.90</td>
<td>23378</td>
<td>29.34</td>
<td>79684</td>
</tr>
<tr>
<td>11</td>
<td>Dutch</td>
<td>47594</td>
<td>59.51</td>
<td>5313</td>
<td>6.64</td>
<td>27073</td>
<td>33.85</td>
<td>79980</td>
</tr>
<tr>
<td>12</td>
<td>British English</td>
<td>17988</td>
<td>22.72</td>
<td>3449</td>
<td>4.36</td>
<td>57746</td>
<td>72.93</td>
<td>79183</td>
</tr>
<tr>
<td>13</td>
<td>Catalan</td>
<td>15214</td>
<td>19.21</td>
<td>1483</td>
<td>1.87</td>
<td>62506</td>
<td>78.92</td>
<td>79203</td>
</tr>
<tr>
<td>14</td>
<td>Slovak</td>
<td>8579</td>
<td>10.84</td>
<td>3523</td>
<td>4.45</td>
<td>67065</td>
<td>84.71</td>
<td>79167</td>
</tr>
<tr>
<td>15</td>
<td>Polish</td>
<td>7713</td>
<td>9.74</td>
<td>1632</td>
<td>2.06</td>
<td>69810</td>
<td>88.19</td>
<td>79155</td>
</tr>
</tbody>
</table>

**Figure 12.** A screen shot of the KDE web page showing the status of translated help documents in the trunk of the l10n repository ordered by language on February 3, 2007.

End users will benefit from a tool that allows them to foresee the availability of help documents in their respective local languages. Information derived from the mined patterns, such as shown in Figure 11b, is an indicator that translated help-documents for a particular language are likely to be soon available. Eleven languages: Italian, Danish,
Swedish, Portuguese, Dutch, Spanish, Russian, German, French, Catalan, and Norwegian Bokmål were found in the evaluation-set. Based on the prediction from our training-set, nine out of these eleven languages with the exception of Catalan and Norwegian Bokmål (82%) were predicted to have updated translations.

To further validate this finding, we consulted the web page that gives the localization status in the trunk of the l10n repository (http://l10n.kde.org/stats/doc/trunk/toplist.php) on February 3, 2007. A screen shot of this web page is shown in Figure 12. Comparing this with the languages we found in our training-set (almost 11 months earlier), it is clear that our approach was able to accurately predict the “top-ten” languages, along with a number of languages further down the list, well in advance.

6.11 Threats to Validity

We discuss the internal and external validities of our approach with regards to the results obtained from its evaluation.

Internal validity refers to addressing the possible factors in our evaluation that bias the obtained results. Our evaluation was on a portion of the history of documents involved in the translation process. Also, the number of mined patterns depends on the user specified minimum support value. In order to assess the impact of the specific period and amount of history, and the user specific minimum support on the accuracy of change prediction, we further validated our approach with different values of these parameters and their combinations. Table 13 shows the coverage, recall, and precision values for different training-sets and evaluation-sets. In the majority of these cases, our
approach has high precision values. In only two cases is the mean precision below 100%. Coverage and recall are reasonable but cannot be considered as particularly high. An interesting observation is at higher values of minimum supports (e.g., 4 and 8), coverage and recall are the same. This indicates candidates suggested by our approach in case of prominent patterns are mostly correct. Also, at lower minimum support values, the difference between coverage and recall cannot be considered high. Improvements in coverage tend to show a decrease in recall and precision. Based on our experimentation results, we can assert that if our approach predicts a change to follow a specific pattern, it is generally precise for a given combination of the above parameters.

External validity refers to addressing the general applicability of our approach and conclusions to any given dataset. We validated our approach on a large open source system (KDE) that actively maintains localized documents in a number of languages. Also, our subject system follows the gnu gettext model with a particular organization and structure of localized documents in the repository. While this system is a prime representative, we do not claim that our results will generalize to any given system (e.g., commercially developed software with a different repository structure.) However, we believe that our mining toolsets are generic enough to be applicable to any translation process with a practice of committing related documents together to a repository.
Table 13. Accuracy of patterns with different history period and mining parameter combinations.

<table>
<thead>
<tr>
<th>Training-set</th>
<th>Evaluation-set</th>
<th>Coverage (%)</th>
<th>Recall (%)</th>
<th>Precision (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Period</td>
<td>MS</td>
<td>Patterns</td>
<td>Period</td>
<td>MS</td>
</tr>
<tr>
<td>2006-01-01 to 2006-02-01</td>
<td>2</td>
<td>139</td>
<td>2006-02-01</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>101</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>58</td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>2006-02-01 to 2006-03-04</td>
<td>2</td>
<td>139</td>
<td>2006-03-04</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>101</td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>2006-03-04 to 2006-04-03</td>
<td>2</td>
<td>139</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>101</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>58</td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>2006-04-03</td>
<td>2</td>
<td>139</td>
<td></td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>101</td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>2006-02-01 to 2006-03-04</td>
<td>2</td>
<td>337</td>
<td>2006-03-04</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>202</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>117</td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>2006-03-04</td>
<td>2</td>
<td>337</td>
<td></td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>202</td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>2006-01-01 to 2006-03-04</td>
<td>2</td>
<td>841</td>
<td>2006-03-04</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>479</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>239</td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>2006-03-04</td>
<td>2</td>
<td>841</td>
<td></td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>479</td>
<td></td>
<td>8</td>
</tr>
</tbody>
</table>


6.12 Related Work

We discuss the work on the analysis of multilingual web sites and web usage data. Also, we briefly discuss approaches utilizing information in source code repositories maintained by tools such as CVS and Subversion with a focus on software changes.

Tonella et al. [Tonella et al. 2001] used a combination of text and structure comparison technique to recover traceability links between multilingual documents. They use a lightweight text comparison approach based on the number of errors produced by applying the UNIX tool spell on a document. The structure comparison is based on the AST and edit distances of web pages. The goal of this work is to support consistency in information provided by a website in multiple languages. Niu et al. [Niu, Strouila, El-Ramly 2002] used sequential-pattern mining to recover usage patterns from the user-session log information of web sites. The goal of this work is to use these patterns to support web site organization to help user-specific navigation and web page recommendation.

Zimmerman et al. [Zimmermann et al. 2004; Zimmermann et al. 2005] used CVS logs for detecting evolutionary coupling between source code entities. They employed sliding window heuristics to estimate the atomic commits (change-sets.) Association-rules based on itemset mining were formed from the change-sets and used for change-prediction. Yang et al. [Ying et al. 2004] used a similar technique for identifying files that frequently change together. Gall et al. [Gall, Hajek, Jazayeri 1998] used window-based heuristics on CVS logs for uncovering logical couplings and change patterns, and
German et al. [German 2004a] for studying characteristics of different types of changes. Hassan et al. [Hassan, Holt 2004] analyzed CVS logs for software-change prediction.

Van Rysselberghe et al. [Van Rysselberghe, Demeyer 2004a] utilized CVS logs in their approach to find frequently applied changes and presented a 2D visualization technique to help recognize change-relevant information [Van Rysselberghe, Demeyer 2004b]. Bieman et al. [Bieman, Andrews, Yang 2003] used logs from software repositories to assist in the computation of metrics for detecting change-prone classes. Burch et al. [Burch, Diehl, Weiβgerber 2005] presented a tool that supports visualization of association rules and sequence rules. However, very little information is provided on how CVS transactions are processed and sequences are mined. Beyer et al. [Beyer, Noack 2005] used the log information in visualizing clusters of frequently occurring co-changes. Dinh-Trong et al. [Dinh-Trong, Bieman 2005] used CVS logs for validating previously developed hypotheses on successful open source development. Chen et al. [Chen et al. 2001] incorporated the CVS commit messages in their source code search tool. El-Ramly et al. [El-Ramly, Stroulia 2004] used sequence mining to detect patterns of user activities from the system-user interaction data. Recently, Kagdi et al. [Kagdi, Collard, Maletic 2007a] surveyed and classified a number of Mining Software Repositories (MSR) approaches for various purposes in the context of software evolution. To our knowledge, uncovering evolutionary dependencies of localized documents with a MSR approach has not been investigated previously.
6.13 Summary

The work presented here applies sequential-pattern mining to uncover frequently co-changed web-documents (i.e., evolutionary dependencies) in the context of internationalization and localization. A large open source system, KDE, was used to evaluate the method. The mined patterns from the historical information in version archives proved to be beneficial for supporting the task of internationalization and localization of web-based documents. This is one of the first published works on the use of sequential-pattern mining for supporting the evolution of documents involved in the natural language translation process.

The performed evaluation shows that our approach accurately predicts the (partially) ordered sets of documents impacted due to a change in a given document, if a similar change pattern had occurred in the history. This is beneficial to developers and translators for the purpose of planning and maintenance as these patterns embody information such as the number of documents, time needed, and the specific order of a proposed change. We envision that our tool could be integrated into a version-control system in the form of commit hooks to support developers and translators. The method also demonstrated to be useful to end users interested in languages that would be better supported in future releases of the system. Our approach accurately predicted, almost a year in advance, which languages would be better supported through updated translations in KDE.

For future work, we plan to further assess our approach on open source systems such as OpenOffice and Apache. Additionally, we are working on extending our
approach to produce finer granularity patterns than the document level and feel that mining patterns at the entry level of PO files should produce even more useful results. Also, translation processes that use different models of translation, such as XLIFF model (www.oasis-open.org/committees/xliff/documents/xliff-specification.htm) for localization, and projects with different organization of translated documents in the repository (e.g., Apache with translated web pages suffixed with the language code) will be investigated to understand the generality of our approach.
CHAPTER 7

Comparing Approaches to Mining Source Code for Call-Usage Patterns

A function-call-usage pattern is a list or set of function calls found in the source code. Although many of these call-usage patterns are very intuitive and may be common knowledge to developers, they are typically not documented. They form latent programming rules that seldom exist outside the minds of developers. Violations of these types of rules are difficult to uncover, report to bug-tracking systems, and fix unless the rules are explicitly documented and made available.

Recently, researchers [Li, Zhou 2005; Livshits, Zimmermann 2005; Michail 2000] have applied data-mining techniques, specifically frequent-pattern mining algorithms, to the problem of uncovering/discovering call-usage patterns from the source code of large systems. The result is a set of rules that describe frequently occurring call-usage patterns within a system. This has been found to be potentially useful for tasks such as identification of standard library/API usages and fault location [Li, Zhou 2005; Livshits, Zimmermann 2005; Michail 2000]. The techniques used are quite efficient; however they sometimes result in a large number of false positives, i.e., reporting potential errors where none actually occur. Large numbers of false positives tend to alienate the users of such tools. But to more accurately identify faults one must apply fairly complex, and computationally expensive, static/dynamic analysis techniques [Livshits, Zimmermann 2005]. On a large system this is not feasible. A data mining or
similar approach can be used to reduce the space to a more reasonable size so that more sophisticated static/dynamic approaches can realistically be applied.

To date, the work using data mining has only used one basic technique, namely *itemset mining*. While this is a very efficient technique, it is not always very accurate and often produces many false positives [Li, Zhou 2005]. This is, in part due to the fact that itemset mining produces patterns that are unordered sets of function calls. So an itemset pattern about the function calls *open* and *close* would allow any ordering of these two calls (e.g., *open(); ... close(); or close(); ... open();*)

To improve on the itemset-mining approach another frequent-pattern mining technique, namely *sequential-pattern mining*, is applied to this problem. Sequential-pattern mining results in a partially ordered list of function calls for the usage pattern. Depending on the domain, the results of sequential-pattern mining tend to be more accurate and have fewer false positives than itemset-mining results. However, sequential-pattern mining comes with a higher computational cost. Here we compare the results of both techniques to ascertain if sequential-pattern mining is a better method to be used for call-usage pattern mining. The comparison is based on accuracy of patterns, candidate patterns and their violations, and computational costs for the version 2.6.14 of Linux kernel. The first issue is very difficult to definitively validate and we use a combination of manual inspection and examination of later versions of the same software system.
7.1 Mining Call-Usage Patterns

A call-usage pattern is a set or list of function calls found in a segment of code. Here, we focus on the call-usage patterns in the function definitions of procedural languages, more specifically C. The goal of the mining process is to uncover call-usage patterns that occur frequently in a software system. The fundamental premise is that frequently-occurring patterns of function calls in a system reflect candidates for standard usages of a library or API. Additionally, if these standard patterns can be automatically reverse engineered then violations of these standard patterns can easily be identified.

To identify violations we must identify variations of frequently-occurring patterns. Specifically, a variant is a proper subset of a frequent pattern that occurs by itself in the systems but in far fewer numbers (e.g., one or two times.) These variants are special cases, or possible errors or misusages. A pair of a variant and a function in which that variant occurs is referred to as a violation.

Mining call-usage patterns from source code can be considered as an instance of the general problem of frequent-pattern mining from any type of data [Agrawal, Srikant 1995]. Before we describe the two approaches, data-mining terminology that is relevant to the discussion is introduced. The input data to frequent-pattern mining algorithms are in the form of transactions (e.g., customer baskets or items checked-out together in market-basket analysis.) Here, an individual transaction corresponds to a single function definition.

The support of a pattern is the number of transactions in which it occurs i.e., the number of functions in which it appears. A frequent pattern has a support at or above
that of a user-specified minimum support in the considered dataset. Such frequent patterns are typically used to form association rules between a pair of patterns (e.g., when pattern $A$ occurs pattern $B$ also occurs.) The confidence of a rule is used to determine the strength of an association rule, and is generally computed from the support of the two patterns to a value in the range $[0, 1.0]$. A high confidence for a rule means the two patterns that make up the rule co-occur in most transactions.

Itemset mining produces patterns that are unordered sets of function calls and we term these unordered patterns. Likewise, sequential-pattern mining produces patterns that are partially ordered lists of function calls so we term these ordered patterns. We now discuss the itemset and sequence mining techniques in more detail.

### 7.2 Itemset Mining

Itemset mining takes a given set of transactions that are composed of some items and finds all the frequently-occurring subsets of items that have at least a user-specified minimum support [Goethals 2005]. Itemset-mining techniques are used in a variety of domains. The most famous application is market-basket analysis for uncovering buying patterns of items frequently purchased together (e.g., beer and diapers frequently bought together.)

Itemset mining performed with a specified minimum support produces a set of candidate unordered patterns from a system. Once such unordered patterns are uncovered, association rules can be generated from them to uncover candidate variants. An association rule is formed from a pair of unordered patterns such that the pattern obtained by their union is also a candidate pattern. The hypothesis is that an association

rule with a very high confidence, but not the highest value of 1.0, is likely to contain a variant. Such an association rule indicates that it has a pattern that occurs by itself in very few transactions.

Itemset-mining approaches have been previously applied for mining call-usage patterns [Li, Zhou 2005; Livshits, Zimmermann 2005]. An approach based on itemset and association-rule mining is taken by Li et al [Li, Zhou 2005] for detecting common programming rules and their variants. They have shown one application of variants in locating potential bugs in a software system.

7.3 **Sequential-Pattern Mining**

Sequential-pattern mining takes a given set of sequences that are composed of items and finds all the frequently occurring subsequences that have at least a user-specified minimum support [Masseglia, Teisseire, Poncelet 2005]. Sequential-pattern mining techniques are typically applied to datasets with temporal or other ordering information. For example, in analyzing market-basket data with the additional timestamp information, patterns such as customers who bought a *camera* are also likely to buy *additional memory* in the next month.

Since function-call usages are inherently ordered, another possible approach is to uncover call-usage patterns with the additional ordering information in the set of calls. Sequential-pattern mining produces a set of candidate ordered patterns with a specified minimum support. Here, the order of calls is determined by their lexical position in the function definition.
However, only partial ordering can be given to calls for which the considered language does not specify a standard order of call evaluations. Both K\&R (Kernighan and Ritchie) and ANSI/ISO C standards leave an inherent order ambiguity of the evaluation of operands and non-deterministic order of function-call argument evaluation. For example, the order of function calls \(a\) and \(b\) in the expression \(a()+b()\) and in the function argument list \((a(), b())\) of the call \(f(a(), b())\) is undetermined. Therefore, compilers take liberty in ordering the calls \(a\) and \(b\) and different compilers assign different ordering. For example, the compiler \texttt{gnu gcc}\ assigns the order \(b, a\) (right to left) and the compiler \texttt{hp aCC}\ assigns the order \(a, b\) (left to right.)

The approach used here is based on the semantics according to the language standards. Therefore, calls involved in constructs such as expression and argument list in situations where there is non-determinism produce a partial ordering. In the example shown in Figure 13, functions \(f2\) and \(f3\) have the same partially ordered pattern \(\{a, c\} \rightarrow \{b\}\) due to non-determinism in the occurrence of calls \(a\) and \(c\) in the expression \(c()+a(.)\). Functions \(f1\) and \(f4\) form completely-ordered patterns. Partially-ordered patterns are quite different from the unordered usage patterns produced by itemset mining. The partially-ordered parts in ordered patterns are non-determinism cases, whereas unordered patterns ignore the ordering information even if it is deterministic.

Once these ordered patterns are uncovered, sequence rules can be generated to uncover variants. A sequence rule is formed between a pair of ordered patterns such that one of them is a (order-preserving) subset of the other. Similar to association rules, a sequence rule with a very high confidence is likely to contain a variant.
7.4 Examples

We first demonstrate the unordered and ordered pattern mining with the help of a synthetic example. Then we give specific examples uncovered from the Linux kernel (v2.6.14.). Consider a hypothetical system with four functions as shown in Figure 13. We will use a minimum support of two for a candidate pattern, i.e., at least two functions must contain the pattern, and a minimum confidence of 0.65 for a variant, i.e., at most 35% of the functions that contain the pattern variant.

Though the functions in the example are incomplete and give very little context, it can be seen that they have similar implementations. We first discuss the unordered patterns and variants produced by itemset mining. Two patterns \{a, b\} and \{a, b, c\} are produced as candidates. The pattern \{a, b\} has a support of four as the calls \(a\) and \(b\) occur together in all four functions, \(f1, f2, f3,\) and \(f4\). Therefore, the association rule \(\{a, b\} \Rightarrow \{c\}\) can be formed from these two patterns with a confidence of 0.75 (i.e., three of the four functions that contain calls \(a\) and \(b\), also contain a call to \(c\).) As this is greater than the specified minimum confidence, the pattern \{a, b\} is reported as a candidate variant. The missing call c that makes the pattern \{a, b\} a variant is only absent in function \(f1\). Therefore the function \(f1\) is reported as a function with the variant \{a, b\} of pattern \{a, b, c\} which causes the pair (\(\{a, b\}, f1\)) to be a candidate violation.

In case of sequential-pattern mining, three ordered patterns \(\{a\} \rightarrow \{b\}, \{c\} \rightarrow \{b\},\) and \(\{a, c\} \rightarrow \{b\}\) are reported as candidates. Pattern \(\{a\} \rightarrow \{b\}\) occurs in functions \(f1\) and \(f2\) where call \(a\) is an argument to call \(b\), and also occurs in functions \(f3\) and \(f4\) where call \(a\) occurs in an earlier expression to the expression that contains call \(b\). This gives it a
support of four. In a similar manner the pattern \{c\}→\{b\} occurs in the function \(f_2, f_3\) and \(f_4\) giving it a support of three. These two patterns are totally ordered, whereas, the third pattern, \{a, c\}→\{b\}, is only partially ordered. This is due to the calls \(a\) and \(c\) occurring in the same expression that is an argument to the call \(b\). This only occurs in functions \(f_2\) and \(f_3\), so the support is two.

\[
\begin{align*}
void \ f_1() \{ \\
d(); \\
b(x+a()); \\
\ldots \\
\}
\end{align*}
\[
\begin{align*}
void \ f_2() \{ \\
\ldots \\
b(c()+a()); \\
k(); \\
\}
\end{align*}
\[
\begin{align*}
void \ f_3() \{ \\
e(); \\
y=c()+a(); \\
b(y); \\
\}
\end{align*}
\[
\begin{align*}
void \ f_4() \{ \\
x=a(); \\
y=c(); \\
b(x+y); \\
\}
\end{align*}
\]

Figure 13. An example of four function definitions with calls to functions \(a, b,\) and \(c\)
for demonstrating patterns, variants, and violations

From these patterns two rules can be formed. The first rule is \(\{a\} \rightarrow \{b\} \Rightarrow \{a, c\} \rightarrow \{b\}\), i.e., when there is a call \(a\) followed by a call \(b\), then a call \(c\) occurs in the same expression as the call \(a\). Of the four functions that contain the pattern \(\{a\} \rightarrow \{b\}\) only two contain the pattern \(\{a, c\} \rightarrow \{b\}\) producing an association rule with a confidence of 0.5. The second rule is \(\{c\} \rightarrow \{b\} \Rightarrow \{a, c\} \rightarrow \{b\}\) with a confidence of 0.67 since the rule applies to two out of the three functions that contain the pattern \(\{c\} \rightarrow \{b\}\). The ordered patterns \(\{a\} \rightarrow \{b\}\) and \(\{c\} \rightarrow \{b\}\) are the only two order-preserving subsets of the ordered pattern \(\{a, c\} \rightarrow \{b\}\) that form sequence rules. However only the second rule satisfies the required
minimum confidence. As a result the ordered patterns \( \{c\} \rightarrow \{b\} \) is reported as a variant in functions \( f4 \).

We have applied both sequential-pattern and itemset mining on the Linux kernel v2.6.14. As examples we will use some of the patterns, variants, and violations uncovered from this system. The unordered pattern \( \{\text{spin\_lock\_irqsave, spin\_unlock\_irqrestore}\} \) occurs in over two thousand functions. This pattern suggests that the calls \( \text{spin\_lock\_irqsave} \) and \( \text{spin\_unlock\_irqrestore} \) are typically used together and is considered to be a candidate usage pattern. However, there are seventeen functions in which the call \( \text{spin\_lock\_irqsave} \) occurs without the call \( \text{spin\_unlock\_irqrestore} \). Therefore, the call \( \text{spin\_lock\_irqsave} \) is reported as a candidate variant. This variant forms seventeen violations. For example the function \( \text{esp\_open} \) in the file \( \text{drivers/char/esp.c} \) contains the violation \( (\text{spin\_lock\_irqsave, drivers/char/esp.c}\#\text{esp\_open}) \) of the pattern \( \{\text{spin\_lock\_irqsave, spin\_unlock\_irqrestore}\} \). This violation indicates that the call \( \text{spin\_unlock\_irqrestore} \) is missing in the function \( \text{esp\_open} \).

The sequential-pattern mining produces the ordered pattern \( \{\text{spin\_lock\_irqsave}\} \rightarrow \{\text{spin\_unlock\_irqrestore}\} \) which occurs the same number of times as the above unordered pattern. This pattern suggests that not only the calls \( \text{spin\_lock\_irqsave} \) and \( \text{spin\_unlock\_irqrestore} \) occur together but they also have a specific order. The sequential violation \( (\text{spin\_lock\_irqsave, drivers/char/esp.c}\#\text{esp\_open}) \) indicates either the call \( \text{spin\_unlock\_irqrestore} \) is missing
(as in the itemset violation) or occurs before the call `{spin_lock_irqsave}` in the function `esp_open`.

### 7.5 Itemset versus Sequence Mining

We now contrast the two approaches with regards to their underlying methodology and characteristics of the uncovered patterns. Both itemset and sequential-pattern mining approaches are driven by their support mechanism for establishing a set of calls in a function as a candidate pattern. In itemset-pattern mining a binary check for presence or absence of all the constituent calls in a function is sufficient to count that function towards its support. The order of calls is completely ignored in mining. In sequential-pattern mining an additional constraint of ordering is required. A function only counts towards the support of a pattern if all the constituent calls are found in the exact same order as in the pattern. Therefore, itemset mining operates on a more relaxed constraint of appearance only than sequential-pattern mining that needs both appearance and order. Itemset mining is a generalized approach, whereas, sequence mining is a specialized approach.

The generalized itemset mining approach affects the coverage with regards to types of patterns and variants. Let us look at each separately.

**Variant multiplicity:** The binary-check approach for counting support ignores the number of times the constituent call is present in a function. If a variant occurs due to multiple call occurrences, they are left uncovered. For example, calls appearing as `lock`, `unlock`, and `lock` with the second call to `lock` missing a matching call to `unlock`. 
**Out-of-Order Variants:** Since the order of calls in a function is completely ignored, variants that occur due to incorrect ordering of calls (e.g., potential bugs or non-standard usage of a call composition) will not be uncovered by itemset mining. Additionally, out-of-order variants may result in false reporting of standard (possibly larger) unordered patterns as order is ignored. This is due to the overgeneralization of multiple variants into a single pattern. The variants \( \{a\} \rightarrow \{b\} \) and \( \{b\} \rightarrow \{a\} \) with support of 5 and 10 respectively would be reported as a subset \( \{ab\} \) with a support of 15.

**Context Information of variants:** Assume that a variant is a true bug due to a missing call(s.) In this case, the only information available to the external user (e.g., a developer or a tool) from a variant pattern is the calls that are missing but not the order in which they should be inserted to fix the bug, or in the case of multiple calls to the same function which particular instances of the calls are part of the pattern.

### 7.6 Comparing the Two Techniques

Ideally one would like to compare the itemset-mining and sequence-mining approaches directly in terms of their effectiveness in solving a particular task. Bug location is one such task that has been previously performed with itemset mining [Li, Zhou 2005]. Unfortunately, such a comparison is not feasible with regards to a large system (e.g., Linux with over 6,000 KLOC) in a reasonable time period. This is primarily due to candidate patterns and variants reported in the order of hundreds or thousands from large software systems. Manual examination of all the candidate patterns is not practical. The lack of documentation of standard usages negates another source to establish a comparison baseline. While the examination of version history is a possible
source of validation, variants may go unnoticed for a number of versions due to their latent nature. They may only begin to be noticed after they have affected the maintainability of the system or a severe bug is discovered. Without a clear comparison there is very little benefit in employing traditional validation metrics such as precision and recall.

One possible approach is a comparison based on the number of patterns, functions with variants, and violations. Frequent-pattern mining produces a large number of patterns from a large-scale software system. Typically, many of the violations derived from these patterns are false positives (e.g., a violation is reported as a potential bug but is not a bug.) A technique that produces much fewer false violations and variants is more desirable. One way to achieve this is to prefer a technique that produces much fewer patterns and thus possibly fewer false variants and violations. However, this approach imposes a risk of discarding valid patterns, i.e., false negatives. A technique that reduces the number of false-positive violations as well reduces the number of false-negative patterns and violations are more desirable with respect to the overall accuracy. These measures also have a direct impact on the number of lines of codes that a developer has to examine and/or an additional analysis tool has to process. This compounded with the false-positive issue, makes these measures reasonable indicators of the effectiveness of an approach.
Table 14. Linux call sequence statistics.

<table>
<thead>
<tr>
<th>System</th>
<th>KLOC</th>
<th>Number of functions</th>
<th>Number of calls</th>
<th>Avg. calls/function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux kernel (v2.6.14)</td>
<td>6,304</td>
<td>112,671</td>
<td>806,297</td>
<td>7.2</td>
</tr>
</tbody>
</table>

Ordered patterns provide more context information to the application or user at the expense of a higher cost. In the case of an itemset mining, the search space of all possible unordered patterns with \( n \) different call usages in a system is \( 2^n \). However, sequential-pattern mining has to potentially consider a combinatorial explosion in the search space of all possible patterns of the order of \( \Theta(2^{nm}) \) with \( m \) partially ordered components each consisting of an average of \( n \) calls. Therefore, a sequential-pattern mining technique could be much more computationally expensive. In practice we found that our implementation of itemset mining took about 62 minutes for a minimum support of 20 (the most time consuming) on the Linux kernel and sequence mining took around 241 minutes. For the higher values both took much less time. Therefore, for lower support values, the cost of sequential pattern mining is approximately four times the cost of itemset mining.

The computation time is really less of an issue as mining will be a relatively infrequent activity compared to inspection of the candidate violations. Therefore, it is the number of variants, functions with variants, and variations that are of serious concern.

7.7 Evaluation on Linux Kernel

In order to facilitate the comparison, we applied both mining techniques on the Linux kernel v2.6.14. First the ordered patterns from all functions were extracted. The
statistics of the considered code base along with the numbers of functions and calls are shown in Table 14.

We developed the tool *callextractor* for extracting call sequences based on the *srcML* format (www.sdml.info/projects/srcml) and the tool *sqminer* for mining frequent patterns. We used *sqminer* for mining the ordered patterns, variants, and violations directly from the ordered patterns extracted by *callextractor*. In addition the tool *sqminer* was configured for itemset mining to mine the unordered patterns, variants, and violations from the unordered patterns formed from the ordered patterns extracted by *callextractor*. Since the results of frequent-pattern mining are sensitive to the externally supplied minimum-support value, six runs of *sqminer* were performed with different minimum-support values and a minimum confidence of 0.9 on a Pentium 4, 3.0GHZ machine with 1GB RAM. The minimum support was doubled in each successive run starting with 20. The number of patterns, variants, functions with variants, and violations are given in Table 15. The following observations can be made:

- Sequence mining found more patterns than itemset mining for all minimum-support values.
- Sequence mining found fewer variants than itemset mining for most minimum-support values.
- Sequence mining found less violations than itemset mining for minimum-support values of 20 and 40, whereas, for minimum-support values 80 160, 320, and 640 the opposite occurs.
The number of patterns and variants decrease with increase in minimum support. The patterns mined with low support values are more likely to be reflective of functions with more specific functionality, whereas, patterns with much higher support may reflect ubiquitous functions that are used throughout the system. The number of variants is lower in the case of ordered patterns in the range of hundreds to thousands for the minimum-support values of 20 and 40. Furthermore, the number of violations is lower for ordered patterns in the range of thousands and as much as 1.5 times. Overall, there are more ordered patterns and fewer variants for sequential-pattern mining than for itemset mining in majority of the minimum support runs. For lower support values of 20 and 40 the number of violations of ordered patterns is less than that of unordered patterns. This suggests that sequential-pattern mining could reduce false positives of variants and violations without compromising false negatives of ordered patterns.

Table 15. A comparison of sequential-pattern mining (ordered) and itemset-pattern mining (unordered) approaches for Linux kernel v2.6.14. Violations are the total number of (variant, function) pairs.

<table>
<thead>
<tr>
<th>Mining</th>
<th>Minimum support</th>
<th>Number of patterns</th>
<th>Number of variants</th>
<th>Number of functions with variants</th>
<th>Violations</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Sequence</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>35832</td>
<td>8907</td>
<td>4652</td>
<td>30284</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>9657</td>
<td>1760</td>
<td>3023</td>
<td>10156</td>
<td></td>
</tr>
<tr>
<td>80</td>
<td>2883</td>
<td>381</td>
<td>2024</td>
<td>4700</td>
<td></td>
</tr>
<tr>
<td>160</td>
<td>996</td>
<td>111</td>
<td>1459</td>
<td>2669</td>
<td></td>
</tr>
<tr>
<td>320</td>
<td>356</td>
<td>37</td>
<td>1018</td>
<td>1558</td>
<td></td>
</tr>
<tr>
<td>640</td>
<td>143</td>
<td>17</td>
<td>754</td>
<td>1089</td>
<td></td>
</tr>
<tr>
<td><strong>Itemset</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>25813</td>
<td>11404</td>
<td>3736</td>
<td>57908</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>7514</td>
<td>2464</td>
<td>2508</td>
<td>15847</td>
<td></td>
</tr>
<tr>
<td>80</td>
<td>2132</td>
<td>421</td>
<td>1697</td>
<td>4093</td>
<td></td>
</tr>
<tr>
<td>160</td>
<td>715</td>
<td>116</td>
<td>1188</td>
<td>2159</td>
<td></td>
</tr>
<tr>
<td>320</td>
<td>254</td>
<td>35</td>
<td>861</td>
<td>1204</td>
<td></td>
</tr>
<tr>
<td>640</td>
<td>106</td>
<td>16</td>
<td>666</td>
<td>840</td>
<td></td>
</tr>
</tbody>
</table>
The ratio of number of variants to the number of patterns gives a general idea as to how many of the patterns are overall violated. Figure 14 shows that this ratio is lower in all cases of ordered patterns. For a minimum-support value of 20, itemset mining reports 44% of the patterns have variants (i.e., 56% of the patterns are followed) and sequential-pattern mining reports 25% of the patterns have variants (i.e., 75% of the patterns are followed.) If variants are used as bug indicators, itemset mining would report more candidate bugs than sequential-pattern mining. As such sequential-pattern mining generally uncovers more potential patterns and reduces the number of variants.

![Variants per pattern chart](image)

**Figure 14.** Variants per pattern showing that sequential-pattern mining outperforms itemset mining as the ratio of the number of variants per pattern is always low for ordered patterns.

Another useful measure is the ratio of the number of violations to the number of functions containing variants. The results of this measure are shown in Figure 15. Once again, sequential-pattern mining outperforms itemset mining for lower support values and performs equally well for higher values. For a minimum support of 20, sequential-
pattern mining would report on average approximately seven violations per function with variants, whereas, itemset-pattern mining would report on average fifteen violations per function with variants, i.e., a ratio more than two times higher. The above results indicate that sequential-pattern mining generally produces a substantially lower number of variants and violations compared to itemset mining.

![Figure 15. Violations per function with variants showing that sequential-pattern mining outperforms itemset mining for lower support values and performs equally for higher support values.](image)

The size of patterns in terms of the number of calls gives us, at least, a cursory idea about the complexity of the typical call usages and its impact on variants (e.g., a larger pattern may mean more possibility of its violation.) Also, the ordering of calls in a pattern may become more desirable with increase in size. For a minimum support of 20, the largest pattern is composed of 16 calls in both itemset and sequential-pattern mining. Overall, itemset mining produces a higher number of larger patterns and a lesser number of smaller patterns than sequential-pattern mining. The singleton and binary patterns
make up approximately 27% and 30% of the total patterns in sequential-pattern and itemset mining respectively.

7.8 Validation

Our interest is in what these variants mean and what tasks they could support. Our first comparison between the two techniques is in regard to a potential bug in an older version of the Linux kernel (v2.6.11) reported by Li et al [Li, Zhou 2005]. The bug in question is the missing call `scsi_scan_host` in the function `sbp2_alloc_device` in the file `sbp2.c` that violates the unordered pattern \{`scsi_host_alloc`, `scsi_add_host`, `scsi_scan_host`\}. This violation is due to the (association) rule \{`scsi_host_alloc`, `scsi_add_host`\} \Rightarrow \{`scsi_scan_host`\} which has a confidence of 0.93. The smaller unordered pattern \{`scsi_host_alloc`, `scsi_add_host`\} occurs in 29 functions while the larger unordered \{`scsi_host_alloc`, `scsi_add_host`, `scsi_scan_host`\} occurs in 27 functions.

A direct comparison of previous itemset-mining results in [Li, Zhou 2005] with that of sequential-pattern mining is not feasible due to lack of insufficient information in reproducing the exact evaluation setup. Therefore, the results of itemset and sequential-pattern mining are compared with regards to the above violation in the v2.6.14. The following results were obtained,

**Itemset mining:** The association rule \{`scsi_host_alloc`, `scsi_add_host`\} \Rightarrow \{`scsi_scan_host`\} was reported with the confidence of 0.83, the unordered pattern \{`scsi_host_alloc`, `scsi_add_host`\} occurs in 42 functions, and the unordered pattern \{`scsi_host_alloc`, `scsi_add_host`, `scsi_scan_host`\} occurs in 35 functions.
**Sequential-pattern mining:** The sequence rule \( \{ \text{scsi\_host\_alloc} \} \rightarrow \{ \text{scsi\_add\_host} \} \Rightarrow \{ \text{scsi\_host\_alloc} \} \rightarrow \{ \text{scsi\_add\_host} \} \rightarrow \{ \text{scsi\_scan\_host} \} \) was reported with the confidence of 0.83, the ordered patterns \( \{ \text{scsi\_host\_alloc} \} \rightarrow \{ \text{scsi\_add\_host} \} \) and \( \{ \text{scsi\_host\_alloc} \} \rightarrow \{ \text{scsi\_add\_host} \} \rightarrow \{ \text{scsi\_scan\_host} \} \) occur in 42 and 35 functions respectively.

Both itemset and sequential-pattern mining are equally likely to report this violation as a bug as they have the same confidence for the rules. However, ordered pattern gives the precise location of where the missing call should be. The corresponding unordered pattern only tells that this call is missing and not the precise location at where it should have been or should be added. Note that implication of association rules in case of itemset mining does not means the order in which calls should occur. It simply tells that whenever the calls \( \text{scsi\_host\_alloc} \) and \( \text{scsi\_add\_host} \) occur, the call \( \text{scsi\_scan\_host} \) should also occur.

We manually inspected the function \( \text{sbp2\_alloc\_device} \) in version 2.6.14 and were not able to confirm the above violation as a bug. So we examined all versions up to version 2.6.16 and found that the call \( \text{scsi\_scan\_host} \) was still absent in the function \( \text{sbp2\_alloc\_device} \). This indicated to us that the above violation is potentially a false positive or it is a bug that has not been fixed yet. In any case, it is safe to surmise that the accuracy of both itemset and sequential-pattern mining is the same.

In order to further examine the results of sequential-pattern mining, we analyzed the functions that were reported to have variants with a minimum support of 20 found by sequence mining but not by itemset mining. Of 1,895 functions with violations, 389
(approximately 20%) were order violations, i.e., all of the calls are present but their composition is not in the right order. Of these order violations, 65 ordered patterns were changed in the later version of the Linux kernel v2.6.16.20. This validates that there are ordering violations that only sequential-pattern mining uncovers.

One example of ordered pattern with an order violation is in the function \texttt{adi\_connect} in the file \texttt{drivers/input/joystick/adi.c} which contains the ordered pattern \{\texttt{gameport\_set\_drvdata}\} \to \{\texttt{kfree}\} \to \{\texttt{gameport\_close}\} \to \{\texttt{kfree}\}. This pattern is violated by multiple variants including the following three rules,

\begin{itemize}
  \item \{\texttt{gameport\_close}\} \to \{\texttt{kfree}\} \Rightarrow \{\texttt{gameport\_close}\} \to \{\texttt{gameport\_set\_drvdata}\} \to \{\texttt{kfree}\},
  \item \{\texttt{gameport\_set\_drvdata}\} \Rightarrow \{\texttt{gameport\_close}\} \to \{\texttt{gameport\_set\_drvdata}\} \to \{\texttt{kfree}\}, and
  \item \{\texttt{gameport\_set\_drvdata}\} \to \{\texttt{kfree}\} \Rightarrow \{\texttt{gameport\_close}\} \to \{\texttt{gameport\_set\_drvdata}\} \to \{\texttt{kfree}\}.
\end{itemize}

In a later version of the Linux kernel (v 2.6.16.20) an additional call \texttt{gameport\_set\_drvdata} was added to this same function creating a new ordered pattern \{\texttt{gameport\_set\_drvdata}\} \to \{\texttt{kfree}\} \to \{\texttt{gameport\_close}\} \to \{\texttt{gameport\_set\_drvdata}\} \to \{\texttt{kfree}\}. As a result, all of the above rules were no longer under violation. This indicates that this violation was a potential bug or a non-standard usage. This example demonstrates that sequential-pattern mining is able to find violations that are not uncovered by itemset mining.
7.9 Related Work

First, we discuss the work related to the problem of finding usage patterns and then the use of frequent-pattern mining methodologies in software engineering for some other purposes. This list is by no means exhaustive but represents a number of different investigations.

Michail [Michail 2000] presented an approach based on itemset and association-rule mining to uncover entities such as components, classes, and functions that occur frequently together in library usages. Similar to the work presented here, Li et al [Li, Zhou 2005] addresses the question of extracting rules and violations of typical usages of function calls in a system. Their approach is based on itemset mining. They show the application of their approach in bug location. Their call extraction uses the gcc front end, whereas, our call-extracting mechanism is based on the language standards and decoupled from a specific compiler implementation.

Livshits and Zimmermann [Livshits, Zimmermann 2005] present an approach based on itemset mining for discovering call-usage patterns from source-code versions. They classified the mined patterns into valid patterns, likely error patterns, and unlikely patterns with additional dynamic analysis. Williams et al [Williams, Hollingsworth 2005b] analyzed usages of function-return values for detecting software bugs via static analysis of a single version and evolutionary changes. A number of researchers used a combination of static and dynamic analyses, and finite state automaton to infer usage patterns and program properties. [Ammons, Bodik, Larus 2002; Nimmer, Ernst 2002; Whaley, Martin, Lam 2002; Yang et al. 2006].
Itemset mining and sequential-pattern mining techniques have been applied to other some problems in software engineering. Zimmermann et al [Zimmermann et al. 2005] used CVS logs for detecting evolutionary coupling between source-code entities. Yang et al [Ying et al. 2004] used a similar technique for identifying files that frequently change together. Burch et al [Burch, Diehl, Weißgerber 2005] presented a tool that supports visualization of association rules and sequence rules. El-Ramly et al [El-Ramly, Stroulia 2004] used sequential-pattern mining to detect patterns of user activities from system-user interaction data. Kagdi et al [Kagdi, Yusuf, Maletic 2006] used sequence mining to extract a sequence of co-changed files from source-code repositories. Xie et al [Xie, Pei 2006] used sequence mining to filter the results of a source-code search tool to report API-usage patterns in which a source-code entity is used. However, a sequence-mining approach has not been used before for function-call usage patterns discovery.

7.10 Summary

We compared itemset and sequential-pattern mining with regards to the number of patterns, variants, and violations when applied to a large system. Our results show that itemset mining produces unordered patterns that are larger in size and higher in support, with more violations when compared with sequential-pattern mining. Itemset mining’s over-generalized behavior results in more false positive candidates than sequential-pattern mining. Sequential-pattern mining produces smaller patterns with the additional benefit of ordering information. We identified candidate violations that were not found via itemset mining. One such case was demonstrated as a part of our validation. The computational cost of sequential-mining is higher than itemset mining. However, this
cost is compensated for the time saved in examining fewer false positives and covering more valid cases.

Comparison metrics as the “gold standard” for validating the violations remains an important and difficult issue. One promising source is the version history as was used in our validation. However, this may not be sufficient due to the latent nature of many of the patterns and their violations that may go unnoticed for a number of versions.

We are currently analyzing call patterns taking into account conditional and iterative constructs. We plan to compare these two techniques on a number of open-source systems, in conjunction with (and without) static and dynamic analysis techniques. We are also extending our call-extraction tool to include other languages such as C++ and Java. A major extension with regards to call extraction in object-oriented languages is the need for analysis of calls via inheritance and polymorphism.
CHAPTER 8

Mining Call-Usage Patterns with Syntactic Context from Source Code

A function-call-usage pattern is an ordered list or set of function calls as they appear in the source code. A common example is an open(); followed by a close();. Although many of these call-usage patterns are quite intuitive and common knowledge, they are typically not well documented. That is, they form latent programming rules that seldom exist outside the minds of developers. Such rules have been found to be potentially useful for tasks such as identification of standard library/API usages and fault location [Li, Zhou 2005; Livshits, Zimmermann 2005; Michail 2000]. Violations of these rules can be very difficult to uncover and fix unless the rules are explicitly documented.

Here, we apply a frequent-pattern mining technique, namely sequential-pattern mining\(^7\), to this problem [Kagdi, Collard, Maletic 2007c]. Sequential-pattern mining results in a partially ordered list of function calls for the usage pattern. Additionally, we add the syntactical context [Kagdi, Collard, Maletic 2007b] in which the calls occur in the source code to these patterns. A good example of where syntactic context is useful is the case of a particular function call, strcat(), always being guarded by another call, strcmp(). The call strcmp() occurring before the call strcat() is necessary, but not

\(^7\) Sequence mining is used synonymously with sequential-pattern mining here.
sufficient. A conditional construct is also required for a correct usage. This additional information is fed into the mining algorithm and reflected in the resulting rules.

In order for developers (or automatic tools) to effectively use rules, or correct violations of a rule, knowing the existence of missing calls is not enough. We also need to know the exact location (e.g., function, statement, or specific call) of the potential violations. Unfortunately, this level of detail with regards to location is missing in the results of previous work based on itemset mining [Li, Zhou 2005]. These techniques are able to give only the specific function where a violation occurs, i.e., the scope of the examination area is the entire function body. This level of granularity can require a daunting amount of manual effort in locating the exact location of the call or its violation in functions with more than a few lines of code (e.g., functions with a 50-100 LOC in a large system such as the Linux kernel is not uncommon.) Our approach identifies not only the rules, but also provides a fine-level granularity with the exact location and the syntactic context of violations of these rules in the source code. For example, the call `strcmp()` is missing from the condition of the `if`-statement guarding the third call to `strcat()` in the function `foo`. This enables developers to quickly assess the potential rule violation in a timelier manner.

To validate the approach, we mined a version of the Apache `httpd` system for function-call-usage rules. The rules our mining approach produced include both the ordering of function calls and their syntactic context in the source code. These rules are used to identify violations in that same version of the system. We then examine a later
version of Apache and determine how many rules are retained and what violations are corrected.

void f1()
  if(d())
    b(x+a());
  ...
}

void f3()
  y=c()+a();
  b(y);
  f();
}

void f4()
  if(d())
    y=a();
    b(CONST+y);
}

Figure 16. An example of four function definitions demonstrating patterns, variants, and violations with the syntactic context information

8.1 Call-Usage Patterns

A call-usage pattern in source code is minimally a list of function calls. Additionally, we include the appropriate ordering or partial orderings of the calls as they appear in the source code and the syntactic context in which each call occurs. The syntactic context of a specific function call is formed by the proximal control structure of the call, such as a surrounding if or while statement. Here, we focus on the call-usage patterns in the function definitions of the C programming language; although almost any procedural language can be supported. The ordering of calls refers to the lexical position with a syntactic construct in source code, and does not necessarily imply a runtime order or control flow of call execution.

Consider the function f1 shown in Figure 16. In this function, the calls a, b, and d occur in a specific order \( \{d\} \rightarrow \{a\} \rightarrow \{b\} \), where the symbol \( \rightarrow \) represents the call order.
In addition, the calls \( a \) and \( b \) are enclosed in an if statement with the call \( d \) in the conditional guard. We add a special notation to the pattern that reflects the syntactic context of each call. The pattern for the function \( f1 \) is represented as:

\[
\{<\text{if}> \text{cond}<\text{if}>\} \rightarrow \{<\text{if cond}="d">\text{a}\ <\text{if cond}="d">\text{b}\ <\text{if cond}="d">\}
\]

In this pattern, the call \( d \) occurs in a condition of an if-statement. It also occurs before (\( \rightarrow \)) the call \( a \). The call \( a \) occurs in the body of an if-statement that has the call \( d \) in the condition, likewise for the call \( b \). The details of this notation and how it is derived from the source code is given in Section 8.2.1.

The goal of the mining approach is to uncover rules of call-usages from the patterns of call-usages found in functions. The basic premise is that these patterns are representatives of the standard usage rules that are prevalent in a software system. For example, the above call-usage pattern of \( f1 \) from Figure 16 also occurs in two other functions namely \( f2 \) and \( f4 \). Once these call-usage rules are discovered then violations of a rule can easily be identified. A violation occurs when a part of a rule is followed, but not the complete rule. Occurrence of calls in a particular pattern in only a single function does not necessarily suggest a call-usage rule. To uncover standard call-usage rules, we need to mine for frequently occurring call-usage patterns in all the functions in the source code. To identify violations of rules, we must also identify sub-patterns of frequently-occurring patterns. Specifically, a variant is an order preserving proper sub-pattern of a frequent pattern that occurs by itself in the system but in far fewer numbers (e.g., one or two times.) A variant may occur due to a missing call, calls made out of order, or a differing syntactic context of a call. For example, the sub-pattern \( \{a\} \rightarrow \{b\} \) is a part of
the above call-usage rule in functions $f1$, $f2$, and $f4$, but occurs by itself in the function $f3$. If this sub-pattern occurs, then there is a probability of 0.75 that the larger, general call-usage rule also occurs. Above a certain (user controlled) threshold this sub-pattern is considered as a variant of the above rule. Note that not all sub-patterns of a rule are variants, as sub-patterns that are always subsumed in a larger pattern (i.e., probability of 1.0) are not reported as variants.

For each variant there is a set of functions where it occurs without the larger, more general rule. A violation is an instance of a variant occurring in a specific function. For example, the variant $\{a\} \rightarrow \{b\}$ in function $f3$ is a possible violation because it is missing the guard call to $d$ that is part of the larger rule. We believe that this representation of a violation assists developers in two complementary ways. First, if a violation is examined and confirmed as a valid misusage in one function, the developer may want to take the same action in other functions with the same violation. Secondly, a developer maintaining, restructuring, or testing a particular function may be interested in all the possible misuses (i.e., variants) that it contains. Based on their negative feedback, this could also assist in quickly confirming a variant as a false positive.

To find these patterns, variants, and violations, we map the problem of mining call-usage rules from source code to an instance of the general problem of frequent-pattern mining from any type of data [Agrawal, Srikant 1995]. Before giving the specifics of our approach, we give the data-mining background that is relevant in our work.
Figure 17. Partially ordered patterns produced by callextractor from functions in Figure 16. An example of four function definitions demonstrating patterns, variants, and violations with the syntactic context information and represented in the form of a transaction.

The input data to frequent-pattern mining algorithms are in the form of transactions. A transaction refers to a group of items that share a common property or occur in the same event (e.g., customer baskets or items checked-out together in the case of market-basket analysis.) The number of transactions in which a pattern occurs is known as its support. If the support of a pattern is at least a user-specified minimum support then it is a frequent pattern in the considered dataset.

Frequent patterns are typically used to form association or sequence rules between pairs of patterns. If the first pattern occurs in a transaction, the relationship between the pair of patterns in a sequence rule could be used to predict the occurrence of the second pattern (e.g., when pattern $A$ occurs, pattern $B$ also occurs.) The confidence (among other metrics such as lift) of a rule is used to determine the strength of an association or sequence rule, and is generally computed from the support of the two patterns to a value in the range $[0, 1.0]$. A high confidence for a rule means the two patterns that make up the rule co-occur in most transactions. Therefore, a rule with a high confidence can be considered as a strong predictor of the second pattern to occur when its first pattern...
occurs. Next, we describe how our approach forms transactions and finds usage patterns, variants, and violations.

8.2 The Approach

Our mining approach consists of two major components: 1) Extracting call occurrences with their ordering information and proximal control constructs from the function definitions in source code; 2) Applying frequent-pattern mining to extract ordered patterns and then use sequence rules to identify call-usage rules and their violations. We now discuss the process of call extraction and mining patterns.

8.2.1 Function-Call Extraction

A prerequisite to mining ordered patterns is the extraction of the call occurrences along with their ordering information and surrounding control constructs, i.e., call-usage patterns, from the functions in the source code. The extraction of calls present in function definitions is a straightforward fact-extraction activity. Our approach is to assign ordering among calls and their proximal syntactic context based on their lexical positions. Calls involved in constructs such as expression and argument lists where there is non-determinism are given partial ordering. In the example in Figure 16, functions $f_2$ and $f_3$ have the same partially ordered pattern $\{a \ c\} \rightarrow \{b\}$ due to non-determinism in the execution order of calls $a$ and $c$ in the expression $c()+a(.)$. Functions $f_1$ and $f_4$ form completely-ordered patterns. The issue of call ordering is further elaborated in [Kagdi, Collard, Maletic 2007c].
The call-extraction tool, namely *callextractor*, is implemented using our srcML platform [Collard, Kagdi, Maletic 2003]. srcML is an XML representation of, primarily C/C++/Java, source code that embeds syntactic information into the text (further details are on www.sdml.info.) In order to perform ordered-pattern extraction, source code is converted to the srcML format using the *src2srcml* translator. The tool *callextractor* reads the srcML file and produces a list of complete, partially ordered patterns, one from each function (i.e., *transaction* in data mining terms.) The patterns include not only the calls, but also the relevant syntactic context. Currently, we support both the *if* and *while* statements, however other structures can be easily supported. Specifically this includes whether a call is in a condition and for which other calls it guards. Because unprocessed source code is used, the calls include both function and macro calls. Figure 17 gives the partially-ordered patterns formed by *callextractor* from the functions shown in Figure 16. Each transaction is uniquely identified by the function name along with its complete file path.

The events in a transaction are identified by a number corresponding to the order position at which the calls occur. For example in Figure 17, the transaction for the function *f1* is identified by the path *path/file1.c#f1*. The function *f1* consists of three events identified with labels 1, 2, and 3. The call to *d*, in the *if*-conditional forms the first event. The calls *a* and *b* represent events 2 and 3 and both are within the body of an *if*-statement with the call to *d*. Similarly, other functions *f2*, *f3*, and *f4* are processed for forming transactions. The only difference is that event 2 in function *f2* and event 1 in
function $f3$ are left unordered due to the language ambiguity in evaluating calls in expressions.

The hierarchical syntactic context is mapped to the list format of the input transactions. The issues encountered during performing this mapping are elaborated in [Kagdi, Collard, Maletic 2007b]. We discuss the representation with the help of the transactions shown in Figure 17. The calls that occur within the body of a control structure are wrapped with the appropriate context. For example, the call $a$ in the function $f1$ for the second event is marked with `<if_cond="d"> a </if_cond="d">`. Secondly, calls in conditions are marked with full elements to show that the call occurs in a condition. For example, the call $d$ in the function $f1$ for the first event is marked as `<if><cond>d</cond></if>`. The same notation is used for both the if and while-statements as both are viewed here as guards to the body.

The tool callextractor uses interface TextReader from libxml2 for processing srcML. This allows a very computationally efficient extraction of ordered patterns. For example, the callextractor takes a little over one minute to process the srcML from over 15,000 source-code files with over 100,000 functions in Linux kernel.

### 8.2.2 Mining Call-Usage Patterns

The specific data mining technique used in our approach is sequential-pattern mining. Sequential-pattern mining takes a given set of sequences that are composed of items and finds all the frequently occurring subsequences that have at least a user-specified minimum support [Masseglia, Teisseire, Poncelet 2005]. Sequential-pattern mining techniques are typically applied to datasets with temporal or other ordering
information. For example, in analyzing market-basket data with the additional timestamp information, patterns such as customers who bought a *camera* are also likely to buy *additional memory* in the next month.

For our call-usage pattern mining, an individual transaction corresponds to a single function definition with its complete call-usage pattern, including the syntactic context. Sequential-pattern mining produces frequently occurring patterns that are partially ordered lists of function calls and the calls syntactic context. Since function-call usages are inherently ordered, our approach uncovers call-usage patterns with the additional ordering information in the set of calls. We term these call-usage patterns as *ordered patterns*. Sequential-pattern mining produces a set of candidate ordered patterns with a specified minimum support.

Once these ordered patterns are uncovered, sequence rules\(^8\) can be generated to uncover variants. A sequence rule is formed between a pair of ordered patterns such that one of them is a (order-preserving) subset of the other. Similar to association rules, a sequence rule with a very high confidence, however not the maximum value of 1.0, is likely to contain a variant. A sequence rule with the value of 1.0 simply suggests that none of the sub-patterns of a pattern are used in isolation, and are always used as a part of a longer pattern.

\(^8\) Sequence rules are not necessarily the same as call-usage rules but are used to differentiate between call-usage rules and their variants.
Both functions with patterns and their variants are recorded and reported by our tools. Additionally, a common pruning mechanism used in frequent-pattern mining is to eliminate all the sub-patterns that have the same support of the corresponding larger pattern. Such patterns are known as closed patterns. Our approach produces only closed patterns. Another benefit of mining closed patterns is the reduction in the number of rules. As the subsets of a pattern that have the same support are pruned, rules are formed only with sub-patterns that have higher support values. The confidence of such rules will be always less than one. Therefore, rules with the confidence value less than one are only formed and examined for inferring candidate variants of both ordered and unordered patterns.

The ordered patterns constructed by the callextractor tool as shown in Figure 17 are fed to our mining tool, sqminer. Mining frequent ordered patterns with sqminer produces a set of closed frequent ordered patterns and sequence pattern rules [Kagdi, Collard, Maletic 2007b; c].

8.2.3 Examples

We first demonstrate the ordered pattern mining with the help of a synthetic example. Then we give specific examples uncovered from Apache httpd (v.2.0.55.) Consider again a hypothetical system with four functions as shown in Figure 16 and their ordered patterns extracted to perform mining as shown in Figure 17. Although the functions in this example are incomplete and give a very little context, it can be seen that they have similar implementations. We will use a minimum support of two for a candidate pattern, i.e., at least two functions must contain the pattern. A minimum
confidence of 0.75 is chosen for a sequence rule, i.e., at most 25% of the functions contain only the pattern variant. Sequential-pattern mining reports three ordered patterns,

1. \{a c\} → \{b\}
2. \{<if><cond>d</cond><if>\} → \{<if_cond="d">a</if_cond="d"> \} → \{<if_cond="d">b</if_cond="d"> \}
3. \{a\} → \{b\}

The first, second, and third patterns occur with a corresponding support value of two, three, and four. The first pattern is partially ordered due to calls appearing in the same expression, whereas the other two are totally ordered. The second pattern contains calls with conditional constructs, whereas, others contain only calls. Here, two sequence rules can be formed from a pair of sub-pattern and a corresponding ordered pattern,

1. \{a\} → \{b\} ⇒ \{<if><cond>d</cond><if>\} → \{<if_cond="d">a</if_cond="d"> \} → \{<if_cond="d">b</if_cond="d"> \} with the confidence of 0.75 and where the function f3 is missing the enclosing conditional call d around the sub-pattern \{a\} → \{b\}
2. \{a\} → \{b\} ⇒ \{a c\} → \{b\} with the confidence of 0.5 and where the functions f1 and f4 are missing the call c.

Only the first rule satisfies the required minimum confidence. As a result the pattern \{a\} → \{b\} is reported as a variant of the pattern \{<if><cond>d</cond><if>\} → \{<if_cond="d">a</if_cond="d"> \} → \{<if_cond="d">b</if_cond="d"> \} in the function f3. Arguably, this case appears to make sense as the implementation of the function f3 is very similar to that of f2. Function f3 is reported as a suspected candidate for future change. So we have the call-usage rule \{<if><cond>d</cond><if>\} → \{<if_cond="d">a</if_cond="d"> \} → \{<if_cond="d">b</if_cond="d"> \} that is obeyed in the functions f1, f2, and f4, the variant \{a\} → \{b\}, and the violation (\{a\} → \{b\}, f3.) In this case, the second rule which contains the pattern \{a c\} → \{b\} is neither reported a rule nor a variant as its sub-pattern \{a\} → \{b\} is declared as a variant by some other pattern. However, this case leads to another interesting issue. Had
the externally controlled value of minimum confidence been set to 0.5, the second rule would also report the sub-pattern \{a\}→\{b\} as a variant of the pattern \{a c\}→\{b\}. Thus, the sub-pattern would have been suggested as a variant of two rules. This forms two additional violations in functions \textit{f1} and \textit{f4}. In our approach, variants and violations are ranked according the confidence of the sequence rule they are formed from. Therefore, the variant and violation of the first rule is reported before that of the second rule.

We have applied sequential-pattern mining on the \textit{Apache httpd} v2.0.55 system. Below are some of the call-usage rules, variants, and violations uncovered from this system with a minimum support of 10 and minimum confidence of 0.9:

1. \{\textit{apr Brigade create}\} → \{\textit{APR BRIGADE INSERT TAIL}\} → \{\textit{ap_pass Brigade}\} occurs in 22 functions
2. \{\textit{apr_socketaddr info get}\} occurs in 10 functions
3. \{\textit{strncasecmp}\} → \{\textit{apr_pstrcat}\} occurs in 12 functions
4. \{\textit{apr file open}\} → \{\textit{apr file close}\} occurs in 27 functions
5. \{\textit{apr Brigade create}\} → \{\textit{apr Brigade create}\} \{\textit{APR_BUCKET IS EOS}\} → \{\textit{apr bucket read}\} occurs in 11 functions

The first pattern does not contain any syntactic context and is composed of calls and macros. The other four patterns include syntactic context. Most of these patterns are self-explanatory and are representatives of common programming practice or idioms. Now, we give two examples of variants and violations.

The variant
\[
\{\textit{<if><cond> ap_xml_parse_input}<\textit{cond}>\} \rightarrow \{\textit{ap_log_rerror}\} \rightarrow \{\textit{dav_push_error}\}
\]
occurs due to the following sequence rule with a confidence of 0.90.

\[
\{\textit{<if><cond> apr_xml_parse_input}<\textit{cond}>\} \rightarrow \{\textit{ap_log_rerror}\} \rightarrow \{\textit{dav_push_error}\} \Rightarrow \\
\{\textit{<if><cond> apr_xml_parse_input}<\textit{cond}>\} \rightarrow \{\textit{ap_log_rerror}\} \rightarrow \{\textit{dav_handle_err}\} \rightarrow \{\textit{dav_push_error}\}.
\]
The variant occurs by itself only in the function `modules/dav/main/mod_dav.c#dav_method_label`, and as a part of the pattern

```
{<if><cond>ap_xml_parse_input<cond><endif>} → {ap_log_rerror} → {dav_handle_err} → {dav_push_error}
```

in every other case. Similarly, the sequence rule

```
{apr_brigade_create} → {apr_bucket_flush_create} ⇒ {apr_brigade_create} → {apr_bucket_flush_create}
→ {APR_BRIGADE_INSERT_TAIL}
```

with a confidence of 0.9 forms another variant and violation of the rule in the function `check_pipeline_flush` located in the file `modules/http/http_request.c`.

Next, we evaluate our approach for the generated rules with the syntactic context.

### Table 16. Apache httpd Call Statistics.

<table>
<thead>
<tr>
<th>System</th>
<th>KLOC</th>
<th>Number of functions</th>
<th>Number of calls</th>
<th>Avg. calls/function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache httpd (v2.0.55)</td>
<td>267</td>
<td>3184</td>
<td>26153</td>
<td>8.21</td>
</tr>
</tbody>
</table>

#### 8.3 Evaluation

Our approach was applied to *Apache httpd (v2.0.55)*. The statistics of the considered code base along with the numbers of functions and calls are shown in Table 16. The tools `callextractor` and `sqminer` were again used to mine call-usage patterns. The minimum-support value was set to ten and the minimum confidence was set to 0.9. With these values, a pattern must occur in at least ten functions to be considered as a candidate call-usage rule, and a subpattern variant must occur by itself in no more than 10% of the functions to be considered as a candidate variant. A Pentium 4, 3.0GHZ machine with 1GB RAM was used to conduct the mining of call-usage patterns. Call-
usage rules as long as seventeen items, and occurring in over six hundred functions were uncovered.

8.3.1 Candidate Call-Usage Rules and Variants

Our first goal is to show that our approach is able to automatically mine call-usage patterns, rules, and their violations. The numbers of candidates mined from Apache httpd (v2.0.55) are presented in Table 17. Both patterns and rules are classified into two different groups: Call and Call+Context. The group Call contains patterns that are entirely composed of calls that have no syntactic context (e.g., \{a\}→\{b\}). The group Call+Context contains patterns that are composed of at least one call with its proximal syntactic context (e.g., \{<if><cond>c</cond></if>\}→\{d\}). Similarly rules are classified in these same two groups. There are over one-thousand rules derived from the mined patterns that occur in at least ten functions. While a large number of rules are devoid of syntactic context, approximately 25% (326 out of 1244) of them contain syntactic context.

Table 17. The number of candidate ordered patterns, rules, variants, and violations in Apache httpd (v2.0.55.) Candidates are categorized based on the inclusion of syntactic context.

<table>
<thead>
<tr>
<th>Category</th>
<th>Patterns</th>
<th>Rules</th>
<th>Variants</th>
<th>Violations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Call</td>
<td>1118</td>
<td>918</td>
<td>112</td>
<td>145</td>
</tr>
<tr>
<td>Call+Context</td>
<td>365</td>
<td>326</td>
<td>28</td>
<td>37</td>
</tr>
<tr>
<td>Overall</td>
<td>1483</td>
<td>1244</td>
<td>140</td>
<td>182</td>
</tr>
</tbody>
</table>
The variants and violations are also similarly classified into the above two groups based on the presence of syntactic context in the corresponding rules. This includes variants that occur due to a missing (or out-of-order) syntactic context. The overall variants in Table 17 are distributed into 80% (112 out of 140) Call and 20% (28 out of 140) Call+Context variants. The number of violations likewise shows a similar distribution in the two groups. The number of violations in both groups (and also overall) is greater than the number of variants. The ratio of violations to variants is 1.29 and 1.32 for the groups Call and Call+Context, and 1.30 overall. Since each variant must occur in at least one function with a violating rule, these ratios indicate that some variants could occur in more than one function. A slightly higher ratio of Call+Context potentially suggests that violating a rule with syntactic context could be more “severe” (spread across a larger number of functions) than other cases. This leads to the next step of evaluation.

Table 18. The number of candidate functions that follow rules and contain variants. These functions are grouped based on the categorization of the rules. An interesting observation is that there is a number of functions containing both categories of rules (and also variants)

<table>
<thead>
<tr>
<th>Category</th>
<th>Functions Containing Rules</th>
<th>Variants</th>
</tr>
</thead>
<tbody>
<tr>
<td>Call</td>
<td>1882</td>
<td>90</td>
</tr>
<tr>
<td>Call+Context</td>
<td>95</td>
<td>18</td>
</tr>
<tr>
<td>Both</td>
<td>573</td>
<td>3</td>
</tr>
<tr>
<td>Overall</td>
<td>2550</td>
<td>111</td>
</tr>
</tbody>
</table>
8.3.2 Using History to Examine Rules

The previous subsection demonstrated that our approach identifies candidate call-usage rules and violations. The next question, and perhaps the more substantial one, is how accurate and useful are these candidates. Validation of these rules is challenging and at times daunting. The biggest piece of the puzzle is a lack of the ground truth or universally accepted gold standard. Manual examination of all the patterns and variants is often not practical as they occur in the order of hundreds or thousands in large software systems. Additionally, many of these usages may require a thorough understanding of a system’s application and implementation domains. An external examiner other than the developers of the system potentially imposes a risk of biasing the results. The lack of documentation of standard usages negates another comparison baseline.

So here, we use the version history for validating our approach. The basic premise is that if the candidate call-usage rules uncovered by our approach are true representatives of standard usages, i.e., global properties, they should also hold true in a more recent version of the same system. Additionally, if the candidate variants and violations uncover true non-standard or improper call usages, they should be corrected (i.e., eliminated or their use should decrease.) Therefore, our validation methodology is to compare the candidates in Table 17 and Table 18 mined from the version 2.0.55 of Apache httpd with those mined with the same mining parameters from a later version of Apache httpd (specifically v2.2.0.) We use the following two measures on Apache httpd to validate our approach for call-pattern rules,
1. **R-R**: The percentage of mined call-usage rules from version 2.0.55 retained in version 2.2.0.

2. **F(R)-F(R)**: The percentage of functions obeying call-usage rules from version 2.0.55 retained in version 2.2.0.

If our approach accurately mines call-usage rules, the values of both R-R and F(R)-F(R) should both be very high percentages. Table 19 shows the comparison of rules mined from versions 2.0.55 and 2.2.0. The column “Common” shows the number of rules that are unchanged between the two versions. The column “Sub-pattern” gives the number of rules in version 2.0.55 that no longer exist as a separate rule because they became subsumed into another rule in version 2.2.0. The column “Super-pattern” gives the number of rules in version 2.0.55 that no longer exist as a separate rule, but at least one of their sub-patterns occurs as a rule in version 2.2.0.

Clearly, the rules in both groups (and overall) are retained across versions as is evident from the R-R values in the range [76%, 82.]. Furthermore, a vast majority of the rules are retained without any change between versions. The results in Table 20 show that the approach also accurately identifies functions that obey rules. Only the functions containing rules that are the same in both versions are shown as these are found to be dominant in Table 19. The fact that a high percentage of the call-usage rules are retained across versions shows that our approach could be used as an indicator of source code that is unlikely to change.
Table 19. Comparison of call-usage rules between v2.0.55 and v2.2.0. The rules are accurately mined as they are retained between versions.

<table>
<thead>
<tr>
<th>Category</th>
<th>Rules</th>
<th>Common</th>
<th>Sub-pattern</th>
<th>Super-pattern</th>
<th>Total</th>
<th>R-R (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Call</td>
<td>918</td>
<td>687</td>
<td>14</td>
<td>15</td>
<td>716</td>
<td>77.9</td>
</tr>
<tr>
<td>Call+Context</td>
<td>326</td>
<td>253</td>
<td>3</td>
<td>10</td>
<td>266</td>
<td>81.5</td>
</tr>
<tr>
<td>Overall</td>
<td>1244</td>
<td>940</td>
<td>11</td>
<td>5</td>
<td>956</td>
<td>76.8</td>
</tr>
</tbody>
</table>

Uncovering call-usage rules and their violations provide a global or system wide property of the standard or state-of-practice in using calls or APIs. Additionally, identifying the functions in the source code, where these rules are obeyed, or not, is important to assist developers in sustaining the changeability and overall evolution of the software system. Table 18 shows the functions that follow the rules, and functions that contain variants of rules. There are many more functions with rules than those with variants of rules. A number of functions contain rules in both groups Call and Call+Context (refer the row labeled “Both” in Table 18.) Rules with syntactic context are found in the least number of functions, whereas, rules devoid of syntactic context are found in the largest number of functions. The same observation holds for functions containing variants.

Overall, this finding of potentially more valid than invalid usages of call-usage rules is expected for a mature system in a major release such as our subject system Apache. However, the most interesting point is that the relatively low numbers of rules that contain syntactic context and their variants indicates that such cases may not be commonly known to developers (or known only to a few experts.) Therefore, we believe
that identification of these “specialized” usages is more valuable than other types of prevalent and ubiquitous rules. Clearly, our approach was able to identify such candidates.

To correctly interpret our validation results with rules becoming sub-patterns and super-patterns between versions, we must note the effects of changes introduced in *Apache httpd* as it evolves from version 2.0.55 to version 2.2.0. These changes could lead to functions being added, deleted, modified, renamed, merged, split and/or relocated to account for such things as feature addition, platform upgrade, and restructuring/refactoring. Usage of APIs calls in a system may be deprecated, replaced, or deleted to align with the changes of evolving APIs themselves. This could affect our candidates in a number of ways. Rules in version 2.2.0 could be smaller (e.g., function deleted or deprecated) or larger (e.g., new function added and now called.) Thus, a rule in one version can become a sub-pattern of a rule in the other.

**Table 20. Comparison of call-usage rules between v2.0.55 and v2.2.0. The functions are accurately identified as they obey rules in both versions.**

<table>
<thead>
<tr>
<th>Category</th>
<th>Functions</th>
<th>Common</th>
<th>F(R)-F(R) (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Call</td>
<td>1882</td>
<td>1517</td>
<td>80.6</td>
</tr>
<tr>
<td>Call+Context</td>
<td>95</td>
<td>58</td>
<td>61.5</td>
</tr>
<tr>
<td>Both</td>
<td>573</td>
<td>419</td>
<td>73.1</td>
</tr>
<tr>
<td>Overall</td>
<td>2550</td>
<td>2039</td>
<td>79.9</td>
</tr>
</tbody>
</table>
For example, the rule

\[
\{\text{if} <\text{cond}> \text{APR\_BUCKET\_IS\_EOS} <\text{cond}<\text{if}>\} \rightarrow \{\text{ap\_pass\_brigade}\}
\]

in version 2.0.55 becomes a sub-pattern of the rule

\[
\{\text{APR\_BRIGADE\_FIRST} \rightarrow \{\text{if} <\text{cond}> \text{APR\_BUCKET\_IS\_EOS} <\text{cond}<\text{if}>\} \rightarrow \{\text{ap\_pass\_brigade}\}
\]

in version 2.2.0. Note that the above rule in version 2.0.55 is not reported as a variant in version 2.2.0. In another example, the rule

\[
\{\text{ap\_log\_error}\} \rightarrow \{\text{clean\_child\_exit}\} \rightarrow \{\text{ap\_log\_error}\}
\]

in version 2.0.55 transforms to the rule \(\{\text{clean\_child\_exit}\} \rightarrow \{\text{ap\_log\_error}\}\) in version 2.2.0 by dropping the first call to an error logging function.

Also, multiple calls in a rule in version 2.0.55 could be replaced by a single new call in version 2.2.0 (e.g., abstracted in a new function.) Similarly, a call in a rule of version 2.0.55 could be replaced by multiple new calls in version 2.2.0 (e.g., pull/delegate refactoring.) In such cases, the “same” rule in both versions may have only a sub-pattern of calls in common or nothing at all. Therefore, a variant in version 2.0.55 may not necessarily be eliminated by its correction according to the recommended rule in version 2.0.55, but by a slightly or totally different new rule in version 2.2.0.

### 8.3.3 Using History to Examine Variants

To show that our approach accurately mines variants and violations of call-usage rules, some of these candidates identified in version 2.0.55 should be observed as corrected to a valid call-usage rule in version 2.2.0. While this may appear to be an obvious and straightforward expectation, variants may go unnoticed for a number of versions due to their latent nature. They may only begin to be noticed after they have
drastically affected the maintainability of the system or a severe bug is discovered. Even if these variants are identified as a valid case of improper use or error, the correction may take a while to appear. For example, as discussed previously in Section 5.2, the violation in version 2.6.11 that Li et al. [Li, Zhou 2005] identified (and received developer’s positive feedback) but was still present in all versions up through 2.6.16. We use the following measures to evaluate our approach for variants and violations.

1. $V-R$: The percentage of identified variants of call-usage rules in version 2.0.55 that transition to call-usage rules in version 2.2.0.

2. $F(V)-F(R)$: The percentage of functions with variants of call-usage rules in version 2.0.55 that transition to function obeying call-usage rules in version 2.2.0.

<p>| Table 21. Comparison of variants of call-usage rules between v2.0.55 and v2.2.0. The results show that our approach correctly identified a number of them. |
| --- | --- | --- | --- | --- | --- | --- |</p>
<table>
<thead>
<tr>
<th>Category</th>
<th>Variants</th>
<th>Common</th>
<th>Un-common</th>
<th>Rule Sub-pattern</th>
<th>New Rule</th>
<th>V-V</th>
</tr>
</thead>
<tbody>
<tr>
<td>Call</td>
<td>112</td>
<td>43</td>
<td>69</td>
<td>11</td>
<td>5</td>
<td>16</td>
</tr>
<tr>
<td>Call+Context</td>
<td>28</td>
<td>15</td>
<td>13</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Overall</td>
<td>140</td>
<td>58</td>
<td>82</td>
<td>11</td>
<td>5</td>
<td>16</td>
</tr>
</tbody>
</table>

Based on the above discussion, we do not expect the values of $V-R$ and $F(V)-F(R)$ to be particular high. However, we did expect some variants to be eliminated from version 2.2.0 as it is a major release. Due to the many-to-many relationship between variants and violations, we compare the variants and the functions containing them between the two versions 2.0.55 and 2.2.0. The rationale for this comparison is that if a variant is confirmed then all its violations (i.e., all functions containing it) can also be
fixed with a similar solution. Therefore, the variants and the functions with variants found in version 2.0.55 should begin to disappear in version 2.2.0, if they are valid.

Table 21 shows the comparison of variants of rules mined from the versions 2.0.55 and 2.2.0. The column “Common” gives the number of variants that are exactly the same between the two versions (i.e., not changed.) The column “Uncommon” gives the number of variants that were reported in version 2.0.55 but are not in version 2.2.0. These uncommon variants are of interest to the validation of our approach. That is, why and how did these variants cease to exist in version 2.2.0? The columns “Rule Sub-pattern” and “New Rule” provide a partial account for their disappearance. The column “Rule Sub-pattern” gives the number of variants of rules in version 2.0.55 that are proper sub-patterns of at least one rule in version 2.2.0. The column “New Rule” gives the number of variants in version 2.0.55 that become rules in version 2.2.0 (possibly due to restructuring or call deletion.) The results show that sixteen such variants are changed to rules in version 2.2.0. This suggests that our approach is able to uncover and report true variants of call-usages that undergo changes. The other variants that are uncounted (66) are likely due to changes such as rename, deletion, or moves to the functions whose calls were part of them, or they are no longer frequent enough for the minimum support value. None of the variants of patterns with structural context were changed to rules in version 2.2.0. This possibly suggests that all 13 of these variants in the category Uncommon are a result of major restructuring. Also, as can be seen a large number of variants (41% overall) are left unaffected (some of them are likely to be false positives.)
We now show a couple of examples of variants in version 2.0.55 that were completely eliminated in version 2.2.0. That is, all functions that had these variants were changed so that they no longer contain the variants. The variant \{apr_brigade_create\}→\{apr_bucket_eos_create\} in version 2.0.55 is corrected in version 2.2.0 with additional macro calls via the rule

\[\{apr\_brigade\_create\} \rightarrow \text{APR\_BRIGADE\_INSERT\_TAIL} \rightarrow \{apr\_bucket\_eos\_create\} \rightarrow \text{APR\_BRIGADE\_INSERT\_TAIL}.\]

The variant with a single call \{apr_bucket_read\} in version 2.0.55 is corrected in version 2.2.0 with the rule \{APR\_BRIGADE\_FIRST\}→\{apr\_bucket\_read\}.

Table 22 shows the functions that are affected due to the sixteen variants in version 2.0.55 becoming new or sub-parts of rules in version 2.2.0. The column "Common" shows the number of unchanged functions with variants that exists in both version 2.0.55 and in version 2.2.0. The column "Uncommon" shows the number of functions with variants in version 2.0.55 that are reported as no longer containing variants in version 2.2.0. The last two columns explain the functions that were fixed due to the corresponding correction of variants in Table 21. Thirty-three functions in version 2.0.55 are found that no longer contain variants due to changes in how they follow the rules. This shows that our approach is able to automatically identify functions that may undergo changes in the next version.
Table 22. The functions with variants in v2.0.55 that are fixed in v2.2.0 due to a variant becoming a rule

<table>
<thead>
<tr>
<th>Category</th>
<th>v2.0.55</th>
<th>Comparison of Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Functions</td>
<td>Common</td>
</tr>
<tr>
<td>Call</td>
<td>90</td>
<td>35</td>
</tr>
<tr>
<td>Call+Context</td>
<td>18</td>
<td>8</td>
</tr>
<tr>
<td>Both</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Overall</td>
<td>111</td>
<td>47</td>
</tr>
</tbody>
</table>

8.4 Threats to Validity

Internal validity refers to addressing the possible factors in our evaluation that bias the results one-way or the other and as such do not represent reality. We did not verify our results with the system’s developers. As previously discussed in Section 8.3.2, the manual effort needed to examine a large number of candidates can be daunting. Additionally as we stated previously, our observation of the results of Li et al. [Li, Zhou 2005] is that this does not imply validation. Even after they got tentative conformation of a bug it was never corrected implying there was actually no bug at all. Bug-tracking dataset was not used for validation due to a (not uncommon) risk that a reported issue (and even confirmed) might not necessarily imply its definite correction. The lack of documentation for standard usages negates another source to establish a comparison baseline. Without a clear comparison there is little benefit in obtaining summarized results with traditional validation metrics such as precision and recall.

We believe that verifying with the actual changes introduced in a later version is a credible source. The mining of candidate rules and their violations is partially sensitive to the parameters minimum support and minimum confidence. Here, we reported and
verified our results for only one set of values. However, we have experienced the same level of performance for various parameter values in this and in our previous work. The calls were extracted from unprocessed source code using lightweight fact extraction. We excluded analysis such as binding of the same name to different functions (i.e., in different translation units.) This exposes the risk of counting two semantically different calls as the same. In any case, we do not claim our approach will always mine, and accurately identify, all possible (and only) rules and their violations. We are examining a number of ranking methods and empirically derived optimal parameter values in this direction.

External validity refers to addressing the general applicability of our approach and conclusions to any given dataset. Two major, but not immediate, versions of a mature system with a very organized development process were selected in this work. This was done to show the scalability and the retention period of the rules. Also, the mining approach is general enough to be applied to other datasets, for example software repositories [Kagdi, Yusuf, Maletic 2006]. However, we do not certainly claim that our approach would produce the same level of results for any arbitrary software system.

8.5 Related Work

First, we discuss the work related to the problem of finding usage patterns. This list is by no means exhaustive, but does represent different investigations.

Michail [Michail 2000] presented an approach based on itemset and association-rule mining to uncover entities such as components, classes, and functions that occur frequently together in library usages. Li et al. [Li, Zhou 2005] addresses the question of
extracting rules and violations of typical usages of function calls in a system. Their approach is based on itemset mining. They show the application of their approach in bug location. Livshits and Zimmermann [Livshits, Zimmermann 2005] present an approach based on itemset mining for discovering call-usage patterns from source-code versions. They classified the mined patterns into valid patterns, likely error patterns, and unlikely patterns with additional dynamic analysis. Wasylkowski et al. [Wasylkowski, Zeller, Lindig 2007] analyzed Java bytecode via control and data flow analyses to obtain pairs of method calls that are directly or indirectly used together on an object. Itemset mining is used to uncover programming patterns, i.e., sets of pairs of method calls (predetermined from bytecode analysis rather than mined) that are frequently used in methods. Also, violations of a pattern, i.e., a missing pair of method calls (and not individual call) are also reported. In our previous work in [Kagdi, Collard, Maletic 2007b] we showed that sequential-pattern mining is able to outperform itemset mining. Williams et al. [Williams, Hollingsworth 2005b] analyzed usages of function-return values for detecting software bugs via static analysis of a single version and evolutionary changes. A number of researchers used a combination of static and dynamic analyses, and finite state automaton to infer usage patterns and program properties [Ammons, Bodik, Larus 2002; Nimmer, Ernst 2002; Whaley, Martin, Lam 2002; Yang et al. 2006]. Kim et al. [Kim et al. 2007] used bug-fix information from software repositories for bug-prediction in source code by using the concept of caching as found in operating systems.

Xie et al. [Xie, Pei 2006] used sequence mining to filter the results of a source-code search tool to report API-usage patterns in which a source-code entity is used. More
recently, Thummalapenta et al., [Thummalapenta, Xie 2007] mined the results returned from a source-code search engine to suggest an intermediate sequence of calls needed to obtain a destination object from a source object. The main goal of this work is to promote reuse from existing examples of API usages. In another work, Acharya et al. [Acharya et al. 2007] used a model checker to generate static traces of user-specified APIs. Difference API usage scenarios are obtained from the static traces that are then reduced to a compact partial-order representation and eventually to usage specifications. However, a sequence-mining approach in conjunction with the syntactic context of calls has not been used in work of others for function-call usage patterns discovery. Kagdi et al. [Kagdi, Collard, Maletic 2007a] give a comprehensive survey of MSR approaches, including those using frequent-pattern mining for various evolution tasks.

8.6 Summary

The approach automatically identifies function call-usage rules, including their syntactic context, violations of those rules, and exact location in the source code of where both the rules and violations occur. The location information is particularly useful for developers to take appropriate (corrective) action. Additionally, it also can reduce the effort developers spend on examining candidate violations. The evaluation of the approach shows that a high percentage of rules are correctly uncovered as they are retained across versions. In addition, a number of identified variants were corrected in a later version. Comparison metrics as the “gold standard” for validating the violations remains an important and difficult issue. One promising source is the version history as was used in our validation. However, this may not be sufficient due to the latent nature
of many of the patterns and their violations that may go unnoticed for a number of versions.

We plan to extend our approach to include other types of syntactic constructs for call context. This includes extending the detection of conditional guards to higher levels than the proximal statements. We are currently extending our call-extraction tool to include other languages such as C++ and Java.
CHAPTER 9
Conclusions and Future Work

The thesis investigated a history-based approach to support software changes and evolution tasks. More specifically, evolutionary couplings reflective of traceability links between source code and other artifacts (e.g., user guides), in source code at fine-granularity levels (e.g., methods, statements, and even comments), and localized document were mined from the change commits stored in the software repositories (e.g., Subversion.) Evaluation results on a number of versions of KDE - a large-scale open system show that these evolutionary couplings were a precise means in supporting/predicting the future changes in software artifacts. Our work also shows that if a history-based approach has a prediction for future changes, it is very likely to be highly precise (i.e., almost no false positives); however it cannot provide prediction for every future change. A hybrid approach that unifies traditional and history-based paradigms was presented in an attempt to improve on the recall (as well as precision.) No other work with the exception of ours has systematically shown the use of version archives in uncovering traceability links, supporting document localization, and source code change prediction at fine-granularity levels over multiple versions.

Furthermore, ours is the only work that demonstrates the use of order and syntactic context of function calls in source code to automatically construct rules that embody latent programming idioms, practice, and function usage. This is a substantial
step forward from any previous work on this topic. Additionally, we showed that the
version archives can be used to devise a systematic and credible method to evaluate the
mined rules and their suggested violations. Such a method was used in our work in
evaluating the rules and their violations on Apache httpd system.

Finally, we hope that the comprehensive survey and taxonomy of MSR
approaches that was conducted during the prologue of this thesis work would serve as a
useful, introductory source to researchers and practitioners interested in the area.

In spite of the progress achieved, our work is far from done in achieving a holistic
change recommendation system that is highly accurate and complete in predicting
changes in source code, but also other artifacts. In this regard, the following two
questions remains of interest in our current and future investigation:

1. How much does the fine-grained expressiveness add to the effectiveness, i.e.,
   accuracy of the prediction rules?

2. What is the additional gain in the effectiveness, i.e., accuracy of the prediction
   rules, with a further integration of source code dependencies from IA?

We conjecture that empirical answers to the above questions would help us to
formulate the laws of software-change prediction. That is, when, for what changes, and
under which contest the history-based paradigm or traditional paradigm, or a unification
of both, is effective and efficient.

An immediate extension of our work is towards additional heuristics for grouping
related change-sets such as textual similarity of commit messages. Also, we are
integrating our tools into IDEs (e.g., Kdevelop and Eclipse) and version-control systems (e.g., Subversion.)

On a final note, the research area of Mining Software Repositories (MSR) has made it possible to redefine the traditional view of Empirical Software Engineering. In that, we now not only use the historical information as a data source for evaluation case studies and experiments, but also to devise process, methods, and tools to actively integrate in the software development/evolution life cycle.
REFERENCES


[Antoniol, Rollo, Venturi 2005] Antoniol, G., Rollo, V. F., and Venturi, G., (2005), "Linear Predictive Coding and Cepstrum Coefficients for Mining Time Variant Information from Software Repositories", in Proceedings of 2nd International Workshop on Mining Software Repositories (MSR'05), St. Louis, Missouri May 17, pp. 74-78


IEEE International Conference on Software Maintenance (ICSM'98), Bethesda, Maryland, November, 16-20, pp. 190-199.


Final Results", in Proceedings of 2nd International Workshop on Mining Software Repositories (MSR'05), St. Louis, Missouri May 17, pp. 58-62.


[Huang, Liu 2005]  Huang, S.-K. and Liu, K.-m., (2005), "Mining Version Histories to Verify the Learning Process of Legitimate Peripheral Participants", in Proceedings of 2nd International Workshop on Mining Software Repositories (MSR'05), St. Louis, Missouri May 17, pp. 84-78.


Workshop on Mining Software Repositories (MSR'05), St. Louis, Missouri May 17, pp. 64-68


[Neamtiu, Foster, Hicks 2005] Neamtiu, I., Foster, J. S., and Hicks, M., (2005), "Understanding Source Code Evolution Using Abstract Syntax Tree Matching", in Proceedings of 2nd International Workshop on Mining Software Repositories (MSR'05), St. Louis, Missouri May 17, pp. 2-6


[Ohba, Gondow 2005] Ohba, M. and Gondow, K., (2005), "Toward Mining Concept Keywords from Identifiers in Large Software Projects", in Proceedings of 2nd International Workshop on Mining Software Repositories (MSR'05), St. Louis, Missouri May 17, pp. 48-52


[Robles, González-Barahona 2005] Robles, G. and González-Barahona, J. M., (2005), "Developer Identification Methods for Integrated Data from Various Sources", in Proceedings of 2nd International Workshop on Mining Software Repositories (MSR'05), St. Louis, Missouri May 17, pp. 106-110


International Conference on Software Engineering Research and Practice
(SERP'03), Las Vegas, Nevada, USA, pp. 448-455.