JOURNAL OF SOFTWARE MAINTENANCE AND EVOLUTION: RESEARCH AND PRACTICE
J. Softw. Maint. Evol.: Res. Pract. 2007; **19**:77–131
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/smr.344

Survey

A survey and taxonomy of approaches for mining software repositories in the context of software evolution

Huzefa Kagdi[1], Michael L. Collard[2] and Jonathan I. Maletic[1,*,†]

[1]*Department of Computer Science, Kent State University, Kent OH, U.S.A.*
[2]*Department of Mathematics and Computer Science, Ashland University, Ashland OH, U.S.A.*

SUMMARY

A comprehensive literature survey on approaches for mining software repositories (MSR) in the context of software evolution is presented. In particular, this 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). The taxonomy is demonstrated to be expressive (i.e., capable of representing a wide spectrum of MSR investigations) and effective (i.e., facilitates similarities and comparisons of MSR investigations). Lastly, a number of open research issues in MSR that require further investigation are identified. Copyright © 2007 John Wiley & Sons, Ltd.

Received 23 August 2006; Revised 29 January 2007; Accepted 1 February 2007

KEY WORDS: software evolution; mining software repositories; multi-version analysis

1. INTRODUCTION

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 (e.g., the Concurrent Versions System (*CVS*)) requirements/bug-tracking systems (e.g., *Bugzilla*), and communication archives (e.g., e-mail).

*Correspondence to: Jonathan I. Maletic, Department of Computer Science, Kent State University, Kent OH 44242-0001, U.S.A.
†E-mail: jmaletic@cs.kent.edu

These repositories hold a wealth of information and provide a unique view of the actual evolutionary path taken to realize a software system. Often these data exist for the entire duration of a project and can represent thousands of versions with years of details about the development. These data include 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.

1.1. Scope

There is a wide range of research investigations that apply mining techniques to software. 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. In addition, 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 for 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.

1.2. Background

Historically, there have been a number of efforts to examine long-term software-project data to better understand software evolution. Lehman *et al.* [1–6] reported various results on the changes in software 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 [1,2,5], metrics of software evolution [6], classification of programs [4], and a theory of software evolution [3]. Weiss and Basili [7] collected and analyzed software data (including changes) from multiple software systems as they were developed. Eick *et al.* [8] observed 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 were 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 the reference model, classification, and/or process model to form a basis for describing the overall MSR investigation in the context of software evolution.

1.3. Previous classifications in MSR

Little effort has been spent on comparing and contrasting MSR approaches. Apart from our own initial survey [9], which examined six approaches, only two other brief surveys have been presented. German *et al.* [10] described 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 focused 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 [11] 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 was described as comparing the elements between two versions (e.g., added, deleted, or renamed lines). They evaluated 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). In addition, we are interested in the types of artifacts (not just source code/programs), types of questions (for various purposes), and methodologies that researchers have investigated in MSR.

In other, less related work, Buckley *et al.* [12] presented 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 were 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.

1.4. 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 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 3 we present our layered taxonomy and its validation (Section 3.2). Section 4 is an in-depth discussion of the published literature in MSR. All of these surveyed approaches are presented with respect to our taxonomy. Section 5 provides a discussion on identified open issues in the MSR research followed by our conclusions.

2. DIMENSIONS OF THE SURVEY

We conducted an initial literature survey [9] 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 Workshops on Mining Software Repositories that took place in 2004, 2005, and 2006 along with a special issue of *IEEE Transactions on Software Engineering* on MSR 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 the ACM/IEEE International Conferences on Automated Software Engineering, Software Engineering, and Software Maintenance 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 as follows.

- 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.

2.1. 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 e-mail 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 [13–18], and to access data from software repositories that are not directly available (e.g., Web page scraping [19] of a defect-tracking system maintained by *Bugzilla*). These approaches are not included in the survey since they only change how data are 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 *CVS* (http://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 [15,20,21]. Also, various branch and merge point detection algorithms have been described [15,22–24]. These issues are important and may have a 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* (http://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* (http://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 e-mail 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.

2.2. 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[‡] question (MBQ) formulated as: if *A* occurs then 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 purpose relates to prevalence questions (PQ). 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.

2.3. 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.

[‡]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’.

For example, metrics for software 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 are directly available from the repositories. This type of metadata are 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 MSR questions to be addressed in a source code aware manner, i.e., in terms of syntax and semantics of the programming languages.

2.4. Evaluation

The realm of open-source development gives us the luxury of publicly available software repositories for many projects. SourceForge (http://www.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[§] is normally used to develop

[§]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.

the models and a later part of history is then used for evaluation. Two assessment metrics, *precision* (i.e., how much of the information found is relevant) and *recall* (i.e., how much of all of 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 [25] 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.

In Section 4 we will see 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.

3. A TAXONOMY OF MSR APPROACHES

The investigations described in the surveyed papers (Section 4) 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.

3.1. A layered taxonomy

In order to see the similarity and differences across the various MSR investigations, we present a layered taxonomy as shown in Figure 1. This taxonomy was developed by identifying the ubiquitous traits found in all of the surveyed literature. Based on the discussion and analyses in Section 4 with regards to the dimensions described in Section 2, 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 2.3, 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.

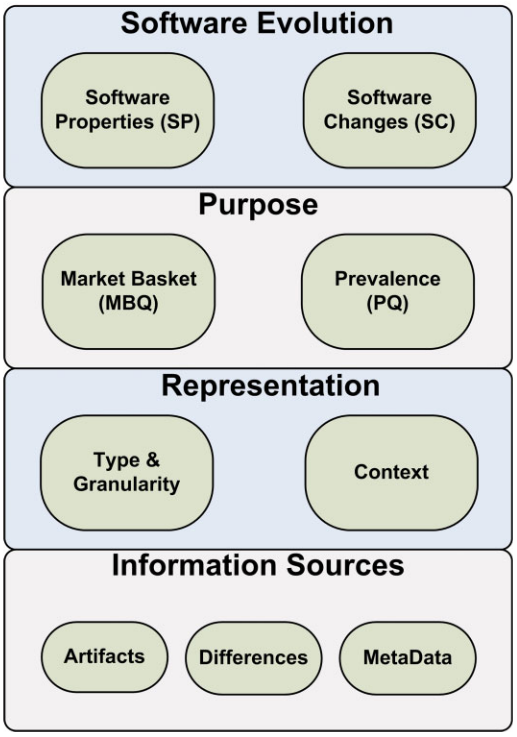

Figure 1. Four-layer taxonomy of MSR approaches.

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 2.1, 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).

3.2. 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 4 with regards to our proposed taxonomy. Tables I–III show the taxonomic description of the considered approaches. The tables are organized by what the approach is studying.

Table I lists approaches studying artifacts, Table II lists approaches studying properties, and Table III 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 I–III. 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 Tables I–III it is clear that all of the approaches have corresponding components in all four layers of the taxonomy, thus showing the expressive quality of the taxonomy.

In Table IV we present the approaches alternatively organized by what evolutionary task they are addressing or studying. We have grouped them into 10 relatively broad categories. For example, all of 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 picture 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 example, Görg and Weißgerber [78] detected refactorings and also provided visualization for their comprehension, and Livshits and Zimmermann [35] 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.

4. MSRS: 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 (subsubsections). 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

Table I. The subset of surveyed approaches that study changes to artifacts. The MSR questions are market-basket questions (MBQ) and prevalence questions (PQ).

Purpose: MSR question	Representation	Information sources	Technique	Task	Approach
MBQ and PQ	Files, classes	Metadata—*CVS*	Metadata analysis		Gall *et al.* [20,26]
MBQ and PQ	Files	Metadata—*CVS* and *Bugzilla*		Logical couplings and change patterns	Fischer *et al.* [23]
MBQ and PQ	Projects, files	Metadata—*CVS* and *Bugzilla*			Fischer *et al.* [27]
MBQ	Files, lines	Metadata—*CVS* and *Bugzilla*		Bug-fixing change analysis	Sliwerski *et al.* [28]
MBQ and PQ	Files, comments	Metadata—*CVS*		Characteristics of different types of changes	German [21]
MBQ and PQ	Files, comments	Metadata—*CVS*		Formalism for querying metadata	Hindle and German [29]
MBQ and PQ	Components of a function declaration	Source code—*CVS*	Source code static analysis	Function interface changes	Kim *et al.* [30]
PQ	Up to statements	Source code	Source code differencing and analysis	Syntactic differencing for fine-grain analysis	Maletic and Collard [31]
MBQ	Files, lines	Metadata, source code—*CVS*	Clone detection methods	Frequently occurring changes	Van Rysselberghe and Demeger [32]
MBQ and PQ	Files, functions, variables context: (Change_Type)	Metadata, source code—*CVS*	Itemset mining	Evolutionary couplings and change predictions	Zimmermann *et al.* [15,33]
MBQ and PQ	Files	Metadata—*CVS*		Change prediction	Ying *et al.* [34]
MBQ	Methods	Metadata, source code—*CVS*		Call-usage patterns	Livshits and Zimmermann [35]
MBQ and PQ	Files	Metadata, source code—*CVS*	Sequential pattern mining	Evolutionary couplings and change predictions	Kagdi *et al.* [36]
MBQ	Files	Metadata—*CVS*	Classification supervised learning	Maintenance relevance relations	Shirabad *et al.* [37–39]

Table II. The subset of surveyed approaches studying changes to properties. The MSR questions are market-basket questions (MBQ) and prevalence questions (PQ).

Purpose: MSR question	Representation	Information sources	Technique	Task	Approach
MBQ and PQ	System, files	Metadata—GNATS, *CVS*, and e-mail	Metadata analysis	Validating hypotheses for successful open-source development	Dinh-Trong and Bieman [40]
MBQ	Files, comments	Metadata—*CVS*, *Bugzilla*, and e-mail source code		Developer identities	Robles and González-Barahona [41]
MBQ	Bugs—context:(metadata)	Metadata—bug-tracking systems		Relationships between bugs/features	Sandusky *et al* [42]
MBQ	Files	Metadata—proprietary repository		Software defects/faults and predictors	Ostrand and Weyuker [43]
MBQ and PQ	Files, lines—context:(filename, lines)	Metadata, source code—*CVS*		Searching and browsing source code	Chen *et al.* [22]
PQ	Packages, files	Versions of a distribution		Evolution of a software distribution	Robles *et al.* [44]
PQ	Function, function calls—context: (file, line)	Source Code—*CVS*	Source code static analysis	Bugs finding and fixing	Williams and Hollingsworth [45,46]
MBQ and PQ	Function calls, parameters, comments, control and assignment statements	Metadata—manuals source code—proprietary repository		Factors for successful software reuse	Selby [47]
PQ	Function calls	Source code—*CVS*		Function usage patterns	Williams and Hollingsworth [48]
MBQ and PQ	Files, classes, methods, parameters	Source code—*CVS*		Incomplete refactorings	Görg and Weißgerber [49]
PQ	Comments	Source code—*CVS*		Communications via source code comments	Ying *et al.* [50]

Table II. (*continued*)

Purpose: MSR question	Representation	Information sources	Technique	Task	Approach
MBQ and PQ	Directories, files	Source code—*CVS*	Software metrics	System complexity	Capiluppi *et al.* [51]
MBQ	System	Metadata—*CVS*		Validation of defect detectors based on static code metrics	Menzies *et al.* [52]
MBQ	Files	Metadata—*CVS*	Visualization	Co-changing files	Van Rysselberghe and Demeyer [53]
MBQ and PQ	Classes	Metadata—*CVS*		Change smells and refactorings	Ratzinger *et al.* [54]
MBQ and PQ	Files	Source code—*CVS*	Clone detection methods	Clones and their relationships	Kim and Notkin [55]
MBQ	Files, lines	Metadata—*CVS* and *Bugzilla*	Information retrieval	Change prediction	Canfora and Cerulo [56,57]
MBQ	Variables	Source code		Concept keywords for program comprehension	Ohba and Gondow [58]
MBQ	Document text	Proprietary repository for requirement traceability		Importance of human guidance	Hayes *et al.* [59]
MBQ	Files, documents	Metadata—*CVS*, *Bugzilla*, and e-mail		New developer assistance	Cubranic *et al.* [60,61]
MBQ and PQ	System	Source Code—*CVS*		Commonly occurring phenomena	Antoniol *et al.* [62]
MBQ	Bug reports	Metadata—*Bugzilla*	Classification supervised learning	Triage bug reports	Anvik *et al.* [63]
MBQ	Directories	Metadata—*CVS*	Social network analysis	Developer roles and contributions	Huang and Liu [64]
MBQ and PQ	Projects	Metadata—sourceforge		Inter-project collaboration	Ohira *et al.* [65,66]

Table III. The subset of surveyed approaches studying changes to both artifacts and properties. The MSR questions are market-basket questions (MBQ) and prevalence questions (PQ).

Purpose: MSR question	Representation	Information sources	Technique	Task	Approach
MBQ and PQ	Files, lines	Metadata—proprietary repository	Metadata analysis	Characteristics of small changes	Purushothaman and Perry [67,68]
MBQ and PQ	File, functions, variables	Metadata, source code—*CVS*		Heuristics for change predictions	Hassan and Holt [69]
PQ	Control structures	Metadata, source code—*CVS*	Source code differencing and analysis	Semantic differencing	Raghavan *et al.* [70]
MBQ and PQ	File, functions	Metadata, source code—*CVS*			Neamtiu *et al.* [71], Godfrey and Zou [72]
MBQ and PQ	System, functions	Metadata—proprietary repository	Software metrics	Complexity of different type of changes	Nikora and Munson [73]
MBQ and PQ	Directories, files, functions	Source code—*CVS*		Types of changes and origin analysis	Tu and Godfrey [74]
MBQ and PQ	Directories, files, functions	Source code—*CVS*	Visualization	Structural and architectural changes	Tu and Godfrey [74]
MBQ and PQ	Subsystems, modules	Source code—*CVS*			Holt and Pak [75]
MBQ and PQ	Subsystems, modules, programs	Metadata—*CVS*			Gall *et al.* [76]
MBQ and PQ	Files	Metadata—proprietary repository	Information retrieval	Classification of MRs based on the cause of a change	Mockus and Votta [77]

Table IV. The approaches surveyed and organized by the MSR tasks they address. The tasks are categorized into related groups.

Evolutionary task category	Approaches
Evolutionary couplings/patterns	Bieman *et al.* [79], Canfora and Cerulo [56,57], Fischer *et al.* [23,27], Gall *et al.* [20,26,76], Hassan and Holt [69], Kagdi *et al.* [36], Shirabad *et al.* [37–39], Williams and Hollingsworth [48], Zimmermann *et al.* [15,33], Ying *et al.* [34]
Change classification/representation	Antoniol *et al.* [62], German [80], Hindle and German [29], Holt and Pak [75], Kim *et al.* [30], Mockus and Votta [77], Nikora and Munson [73]
Change comprehension	Beyer and Noack [81], Burch *et al.* [82], Chen *et al.* [22], Chen *et al.* [83], Cubranic *et al.* [60,61], Gall *et al.* [76], Görg and Weißgerber [78], Hindle and German [29], Holt and Pak [75], Kim *et al.* [84], Purushothaman and Perry [67,68], Claudio [85], Robles *et al.* [44], Van Rysselberghe and Demeyer [53], Venolia [86]
Defect classification and analysis	Anvik *et al.* [63], German [21], Livshits and Zimmermann [35], Menzies *et al.* [52], Nagappan *et al.* [87], Ostrand and Weyuker [43], Sandusky *et al.* [42], Sliwerski *et al.* [28], Williams and Hollingsworth [45,46]
Source code differencing	Maletic and Collard [31], Neamtiu *et al.* [71], Raghavan *et al.* [70], Sager *et al.* [88]
Origin analysis and refactoring	Dig *et al.* [89,90], Godfrey *et al.* [72,91], Görg and Weißgerber [49,78], Henkel and Diwan [92], Kimand Notkin [55], Ratzinger *et al.* [54], Tu and Godfrey [74], Weißgerber and Diehl [93], Zou and Godfrey [24]
Software reuse	Selby [47], Van Rysselberghe and Demeyer [32], Xie and Pei [94]
Development process and communication	Dinh-Trong and Bieman [40], El-Ramly and Stroulia [95], Hayes *et al.* [59], Huang and Liu [64], Mockus *et al.* [96], Ohba and Gondow [58], Ohira *et al.* [65,66], Ying *et al.* [50]
Contribution analysis	Koch and Schneider [97], Mockus *et al.* [96], Robles *et al.* [41,98]
Evolution metrics	Capiluppi *et al.* [51], Godfrey *et al.* [72,91], Menzies *et al.* [52], Nagappan *et al.* [87], Nikora and Munson [73], Tu and Godfrey [74]

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.

4.1. 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 the 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 those 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 section, 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.

4.1.1. Logical couplings and change patterns

In the work presented by Gall *et al.* [20,26], common semantic (logical and hidden) dependencies between classes due to addition or modification of a particular class were 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 was recorded (e.g., class $A = \langle 1, 3, 7, 9 \rangle$). The classes that changed in the same release were 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 were inferred to have dependencies.

This technique was 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 suggested 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* were integrated into a *SQL* database [23]. These data were used to trace the origin and modifications of files in the evolution of a system. A heuristic-based merge-point identification algorithm was 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 the *Mozilla* project as found on 14 December 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 was 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.) was 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 of 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 [27]. A case study was 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 3800 to 7000. 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., *FreeBSD CVS* log records were analyzed for keywords *netbsd*, *openbsd*, and *linux*). The distribution of this information was studied from the period 1994–2004. The results indicate that *OpenBSD* continues to be more decoupled from the rest of the projects.

4.1.2. Heuristics for change predictions

Hassan and Holt [69] used 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 were 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 stated 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 were 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 [69] (specifically the author's Table III) 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 gave better values.

4.1.3. Bug-fixing change analysis

A combination of information in the *CVS* log file (change deltas) and *Bugzilla* was used to study *fix-inducing* changes, i.e., new changes that were introduced to fix an earlier reported problem, by Sliwerski *et al.* [28]. 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 were used to determine the *CVS* deltas that are related to a change that fixes a bug. Given such a change (fix), the modified lines were identified by using the *cvs diff* command. The latest deltas that also affect the involved lines were found using the *cvs annotate* command. These deltas were further analyzed by heuristics to filter out false positives. The remaining 'true' deltas were considered to be the changes that induced a given fix.

The main question investigated was 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 was 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 was 3.82. Overall, the fix-inducing transactions were about three times larger than the non-fix inducing transactions. A high risk of introducing fix-inducing changes was found on Saturdays and Fridays for *Eclipse* and *Mozilla*, respectively.

4.1.4. Characteristics of different types of changes

German [80] examined software repositories (*CVS*) to study the evolution of the e-mail client *Evolution* between 1998 and 2003. The *CVS* annotations were 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 [21] the focus was on studying different types of MRs. The following are a set of representative questions that were 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 of 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 MR. The number of functions added or deleted with bugMRs was less overall than any other type of MR. The analysis of 3094 MRs from the year 2002 found 2261 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.

4.1.5. Formalism for querying metadata

Metadata, such as those found in the *CVS* log files, were modeled using a graph representation by Hindle and German [29]. A query language based on first-order and temporal logic, namely *SCQL*, was defined to facilitate questions on changes at a level expressed by the metadata. The approach was 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 4748 files, and between 1398 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.

4.1.6. Characteristics of small changes

The focus of a study presented by Purushothaman and Perry [67,68] was to understand the impact of small changes, particularly one-line changes, with regards to faults, the relationship between different types of changes (i.e., add, delete, and modify), the reason for the change (i.e., corrective, adaptive, and perfective), and dependencies between changes. A change was 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 were 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 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 [77] were 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.

4.1.7. 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 was presented by Chen *et al.* [22]. A historical context for all of the lines in the latest version of a system was formed by associating each line with all of 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 was specified in terms of keywords (e.g., *login*). The tool displays all of 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 approximately range between 24 KLOC and 49 KLOC and average between 10.8 and 38.3 revisions per 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. When deciding whether *CVS* comments are better than *grep* or *vice versa* with *p*-values, overall *CVS* comments did better than *grep* for all of the applications. However, the results were inconclusive for individual applications.

4.1.8. Successful open-source development

The work by Dinh-Trong and Bieman [40] is an external validation of five of the seven hypotheses pertaining to successful open-source software development given by Mockus *et al.* [96] 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 [96] 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 e-mail archives (sent to freebsd-bugs@FreeBSD.ORG), a bug database (*GNATS*), and a *CVS* repository (log file). Only problem reports (PRs) 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 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 e-mail 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.* [96] 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% of the new functionality. The group will represent less than 25% 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.

4.1.9. Developer identities

It has been observed [96–98] 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, e-mail address in the project mailing lists and bug-tracking system). Robles and González-Barahona [41] presented a methodology based on heuristics such as spatial locality (e.g., in source-header comments, an e-mail address and a developer's name occur together) on the identities collected from various repositories (e.g., e-mail archives, *CVS* repositories, *Bugzilla*). The approach was validated on the *GNOME* project. The examined data set consisted of 464 953 e-mail messages from 36 399 distinct e-mail addresses, 123 739 bug reports from 41 835 reporters, 382 271 comments from 10 257 posters, and approximately 2 000 000 *CVS* commits by 1067 committers. The results indicate that these identities actually correspond to 34 648 unique persons. The authors further plan to investigate gender and nationality distribution.

4.1.10. Relationships between bugs/features

An examination of the bug reports (BRs) from a bug-tracking system was 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 by Sandusky *et al.* [42]. A bug-tracking system typically contains a category field that explicitly stores the relationship (if any) of a BR 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 the resolution of other bug fixes. Moreover, BRs 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 BR (e.g., texts such as '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 of the relationships are still manually discovered and maintained. A random set of 385 bug reports was selected from a population of approximately 182 000 BRs 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 BR. Duplications accounted for 43% and dependencies accounted for 19%, with 33% attributed to informal relationships.

4.1.11. Software defects/faults and predictors

Ostrand and Weyuker [43] used 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 lines of code, age in versions, number of faults in a previous version, and source code language were considered as independent variables. The identification of MRs that represents faults/defects were performed by examining the roles of, or interviewing, the reporters. The MRs reported by testers were considered as strong candidates for faults, whereas those reported by developers require further inspection. The goal of this tool was 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.

4.1.12. Evolution of a software distribution

Robles *et al.* [44] studied 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 was 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 were 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/) was used to process this file and help compute the above measures. This study reported the following observations.

- The overall size of Debian (of the order of millions of lines of code) approximately doubled every two years.
- There are relatively fewer large packages (over 100 KLOC) than small packages (1 KLOC to 50 KLOC) in all of 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 decreased 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.

4.1.13. Completeness of ChangeLog files

Chen *et al.* [83] examined 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 whether *ChangeLog* records 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* were 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) that 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) that 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) that 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.

4.2. 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 to examine 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.

4.2.1. Bug finding and fixing

In an approach presented by Williams and Hollingsworth [45,46], bug-fix information was automatically mined from the source code repository to improve bug finding/fixing tools. The type of bug considered was a function-return-value check. The existence of this type of bug in the considered systems (*Apache* and *Wine*) was determined by manual inspection of the source code repository. A custom tool for detecting function-return-value checks was developed based on a traditional compiler-like parser. The tool combs through all of 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 a historical context (also known as history-aware ranking) are given a higher priority than those obtained by the contemporary (also known as 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 of 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 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 history-aware ranking in terms of precision. However, overall there was a better precision for the history-aware 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 (history-aware ranking), the recall for the warnings in the lower half (naive ranking) is also improved.

4.2.2. Factors for successful software reuse

The study presented by Selby [47] was an investigation of the factors that characterize successful software reuse in large-scale systems. Both design and implementation factors characterizing successful software reuse were 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 was conducted and evaluated based on the goal–question–metric (GQM) paradigm. The goals were set, questions were devised to fulfill each goal, and metrics were 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 non-parametric ANOVA (analysis-of-variance) model. The data were 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. A 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.

4.2.3. Function usage patterns

A method to automatically detect function usage patterns was presented by Williams and Hollingsworth [48]. Specifically considered were 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 was to find new instances in the current version. A C parser was used to identify function calls. Instances were 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 was considered to follow the function usage pattern. The tool was applied to the software repository of the *Wine* project. Overall, about 50 million instances were reported with 2175 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 [48, Tables 1 and 2].

4.2.4. *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 was described by Görg and Weißgerber [49]. 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 was used to obtain all of 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 were obtained. Further analysis of these lists resulted 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 were further analyzed to determine inconsistencies that are 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. Note 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*, was described. The approach reported five (two methods in sub-classes and three methods in sibling-classes) and seven (three methods in sub-classes 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.

4.2.5. *Function-interface changes*

A fine-grain analysis and classification of function-signature changes was presented by Kim *et al.* [30]. A fact-extraction tool developed by the authors was used to identify function signatures present in all versions. The obtained function signatures were 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 were 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 were 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 [30, Table 1] 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, was presented. The authors reported that in case of the *Subversion* project, 77% of functions were never involved in a signature change, and 95% of the functions 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 could foresee the application of this in predicting future changes in a function signature.

4.2.6. Communication via source code comments

Ying *et al.* [50] presented 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 9 February 2005 was described. It consists of 2213 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 showed that a task comment may include an author’s identity and change-request identifiers that may be useful for MSR applications.

4.3. Source code differencing and analysis

Source code repositories contain differences between versions of source code (i.e., difference data). As discussed in Section 2.1, these difference data are 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.

4.3.1. Semantic differencing

The tool *Dex* was presented by Raghavan *et al.* [70] for detecting syntactic and semantic changes from a version history of C code. All of the changes in a patch were considered to be part of a single higher-level change, e.g., bug fix. Each version was converted into an abstract semantic graph (ASG) representation. A top-down or bottom-up heuristics-based differencing algorithm was 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 were 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*. The differencing algorithm takes polynomial time to the number of nodes. Average times of 60 seconds and 5 minutes per file were reported for *Apache* and *GCC*, respectively, on a 1.8 GHz Pentium IV Xeon 1 GB RAM machine. The six frequently occurring bug-fix changes as a percentage of patches in which they appear were reported. *Dex* reported that 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.* [71] used a partial abstract syntax tree (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 were the entities of interest. The differencing was 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 posed 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 that *Vsftp* and *Apache* are more suitable for dynamic software updating while *OpenSSH* involves a risk.

4.3.2. Syntactic differencing for fine-grain analyses

Maletic and Collard [31] presented 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 the following.

- 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. In addition, 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.

4.3.3. Identification of refactorings in changes

Weißergerber and Diehl [93] presented a technique for identifying changes that are refactorings. The line-based differences of files in a *CVS* commit were 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 were 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* were used to examine whether refactorings caused less bugs than other changes. A change was considered to introduce a bug if a BR was 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, were 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 1707, respectively. It was found that a high number of refactorings per day followed by no increase in the number of bugs per day was prevalent in most periods of history, an indication that refactorings are less bug prone. However, they also found instances where a high number of refactorings per day was followed by an increase in the ratio of bugs to days.

Dig *et al.* [89] 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 was first performed via a lightweight AST to identify source code entities along with their fully qualified names. Then an information retrieval technique, namely Shingles encoding, was 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 were analyzed. If they continue to have similar calls in both versions, a candidate refactoring is confirmed as a true case. An *Eclipse* plugin was 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* were used for evaluation. Refactorings were identified with both precision and recall of over 85%. Results of manual examination from previous results were used for validation. The end goal of this work was to support automatic replaying of refactorings performed in a component to the clients of that component. Henkel and Diwan [92] have a similar goal but record refactorings as they are performed by a component/library developer in the *Eclipse* IDE. This method was realized as a plugin within the *Eclipse* IDE.

4.3.4. Changes in micro patterns

Kim *et al.* [84] studied the changes in micro patterns. A micro pattern is a programming idiom for a class in Java [99]. The interest was 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 was to correlate the changes in the micro patterns with the reported bugs. The *CVS* repositories of three open-source projects *ArgoUML* (1262 versions), *Columbia* (1652 versions), and *jEdit* (1449 versions) were used. The coverage of the different types of micro patterns seen in the considered latest versions of *ArgoUML*, *Columbia*, and *jEdit* were 55%, 79%, and 81%, respectively. The changes in the type of a micro pattern in *ArgoUML*, *Columbia*, and *jEdit* were 6%, 5%, and 4.1%, respectively, of the total micro-pattern changes. The top 20 frequently changed micro patterns and top 20 bug-prone micro patterns were listed for all three considered projects. There was 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 noted that the correlation between the number of changes in the micro patterns and the introduction of bugs remains inconclusive.

4.3.5. Detecting similar Java classes

Sager *et al.* [88] 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 ASTs were generated from two versions of the source code and then converted to FAMIX [100] trees for language-independent representation. The two FAMIX trees corresponding to the two versions of source code were compared with the tree algorithms. The approach was evaluated on the *Eclipse compare plugin* (versions 3.0 and 3.1) using the tree-edit distance algorithm (proved to be 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 were shown in the form of a *heatmap*, a two-dimensional plot with a box used to show the similarity. For details we refer the reader to [88, Figure 6].

4.3.6. Studying API changes

Dig and Johnson [90] studied the API changes between two versions of a framework/library (referred to here as a component). The interest was in the classification of the API changes as *breaking* and *non-breaking* changes. An API change of a component was 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 was 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 was 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 were used to conduct the investigation. Two versions of each system were 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.

4.4. Software metrics

Software metrics are used to quantitatively assess various aspects of software products, projects, and processes. 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.

4.4.1. Complexity of different changes

The work presented by Nikora and Munson [73] 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 were 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 12 metrics in proportion to the amount of unique variation contributed by that complexity metric.

A case study on the *Mission Data System* (MDS) was described. MDS is a system developed by the California Institute of Technology's Jet Propulsion Laboratory (JPL) operated as a NASA laboratory. The history of MDS consists of 1500 builds, 65 000 versions, and more than 15 000 functions. The hypothesis here was that not all of the changes contribute equally to the overall complexity of the system: changes to comments could be simple while others may have a substantial impact on the structure of software modules. The goal was 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 1400 problem reports.

The results indicate that not all of the builds were equivalent. The control structure of the system changes much more rapidly than others, and a substantial number of changes are attributed to it. There is a fluctuation of change activities in all of 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.

4.4.2. Change-prone classes and change-couplings

In another approach by Bieman *et al.* [79], a metrics-based approach was 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 was used in understanding these clusters of classes. The following research questions were investigated.

- Is it possible to identify and visualize the most change-prone collection of classes in an object-oriented 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 can the change-prone information be visualized?

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

A case study was described on an industrial application written in C++. Two versions, identified as A (first stable version) and B (latest version), were considered. Version A consists of 199 classes and 24 KLOC. Version B consists of 227 classes with about 32 KLOC. There were 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 notation. The visualization revealed that there were change-coupling relationships between classes that were not represented by any design relationships. The class-level metrics revealed 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.

4.4.3. Types of changes and origin analysis

The tool *Beagle* (also discussed in Section 4.5) 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 [74]. Evolution metrics lines of code, S-complexity, cyclomatic complexity, and number of function parameters, etc., were 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 was represented by the Euclidian distance between their vectors. The similarity values were used for origin analysis of a given entity, i.e., the lesser the distance, the greater 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 was 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* was also discussed. This technique was 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 was 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*.

4.4.4. System complexity

Capiluppi *et al.* [51] presented an approach studying the complexity of a software system. The complexity was 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 was 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 was described on the *ARLA* system, an implementation of the AFS distributed file system. The latest release consists of 150 KLOC and overall 45 developers participated in this project. The results showed 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 on 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).

4.4.5. Validation of defect detectors

The metrics and defect data available from the NASA's Metrics Data Program (MDP) (collected for almost eight years in some cases) were utilized by Menzies *et al.* [52] 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 described their study on five NASA applications. Various data-mining tools LSR, M5, J48, and ROCKY were used to automatically generate defect detectors. The results obtained from these data miners were compared using the DELPHI approach (i.e., human-experts view). Assessment measures such as precision, recall, accuracy, and effort in terms of lines of code 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 lifecycle by using data from the local history. Furthermore, the authors suggest the use of static-code metrics as secondary indicators.

4.4.6. Predicting post-release failures

Nagappan *et al.* [87] 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 used five Microsoft products to validate the following four hypotheses, which 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.

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 built via PCA to predict post-release defects. Hypothesis (H2) was rejected. Hypothesis (H4) was partially supported. Hypothesis (H4) was only supported for projects that have the same or similar defect distribution. The authors advise caution in using 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.

4.5. 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 [101]. 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.

4.5.1. Co-changing files

Van Rysselberghe and Demeyer [53] proposed 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 was 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*, was also described. The found visual patterns indicative of the above aspects were 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 was presented by Beyer and Noack in [81]. The co-changes derived from the log files of a version-control system were represented as an undirected bipartite graph. This graph was termed a *co-change graph*. An undirected edge was drawn from an artifact node to a transaction node if an artifact was 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, was used to layout the co-change graph. A formalism was 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 small number of transactions. The artifacts that are placed together in the layout give an impression of a cluster. The approach was 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 were presented in a table in [81]. 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 with the authoritative decomposition (e.g., subsystems) of the system. Nodes (i.e., files) belonging to the same subsystems were given the same color. In conclusion, most of the clusters in the layout conformed 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.

4.5.2. Structural and architectural changes

The tool *Beagle* provides two simultaneous views referred to as structure diagrams and dependency diagrams [74]. 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 [74]. For an instance of a view showing a structural diagram and dependency diagram, we refer the reader to [74, Figure 5].

Holt and Pak [75] presented a visualization tool, namely *GASE*, for representing software structural changes. The software system was 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 were used to show the differences in the nodes and edges between two versions of a software system. The tool was applied on 11 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 indicated 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 of these observations were verified with the developers of the subject system.

Gall *et al.* [76] presented 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) was 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 between 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). A color spectrum (i.e., a customized rainbow scale) is used to choose (successive) colors equal to 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 was demonstrated on 20 releases of an industrial system over a period of two years. The subsystem system consisted of eight subsystems, 47–50 modules, and 1500–2300 programs. For further information on the various views and observations reported on the subject system, we refer the reader to [76].

4.5.3. *Change smells and refactorings*

A graph visualization with nodes representing classes and edges representing logical couplings was used to identify change smells by Ratzinger *et al.* [54]. 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 [102]. 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*, were discussed. Standard refactorings such as move method and move field were suggested to alleviate the man-in-the-middle problem. Refactorings such as move method and extract method were suggested to improve the code exhibiting data-container smell. A case study was 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 discussed 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 were detected with the technique described in [49] were visualized in Görg and Weißgerber [78]. 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) were demonstrated on the *jEdit* and *Tomcat* projects. The visualization provides class-hierarchy and package-layout views. Different colors were used for representing different kinds of refactorings. UML symbols were used for representing both classes and the relationships (e.g., generalization). The relationship symbols are appropriately colored to indicate their part in the corresponding structural refactorings.

4.5.4. Visualizing data mining rules

Burch *et al.* [82] discussed techniques to interactively visualize association and sequence rules mined from software archives by using data-mining techniques. They further presented 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 were realized in the tool *EPOSee*. Examples of these views were demonstrated on the *Mozilla* repository. Clusters and outliers of changed files identified in these examples were discussed.

4.6. 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 address 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 ASTs and program dependency graphs (PDGs). In this section, we discuss the 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).

4.6.1. Clones and their relationships

An approach based on the history of source code clones was presented by Kim and Notkin [55] to assist in maintenance. A clone-detection tool, namely *CCFinder* (http://www.ccfinder.net), was used to identify clone groups in each version of a program in the *CVS* repository. A cloning relationship was assigned to the corresponding clone groups in the consecutive versions. The cloning relation was 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 were 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 was 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 of 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 reader to [103]. The investigation of the above questions was 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'.

During 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 was 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 systems found that about 25–48% of the clone lineages were actually diverged (possibly due to refactorings) from their original place (group) to some other location (group). Therefore, they are not completely eliminated.

4.6.2. Frequently occurring changes

The concept of frequently applied changes (FACs) was introduced by Van Rysselberghe and Demeyer [32]. FACs were defined as changes occurring multiple times in the version history of a system. All of the *CVS* deltas were examined (via *cvs log* command) and their corresponding source code changes (via *cvs diff* command) were recorded in a text file. A clone-detection tool, *CCFinder*, using parameterized token matching was applied to this text file to find similar pairs of source code changes (i.e., clones). The *CVS* deltas corresponding to these clones were considered as the FACs. This technique was 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 were 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 suggested 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 suggested such FACs may be used to identify recurring change patterns and, in turn, identification of refactorings.

4.6.3. Code duplication and origin analysis

In another approach by Godfrey *et al.* [91], both parameterized and metrics-based string-comparison techniques were used to study code duplication within the file-system component of the *Linux* operating system. In addition, 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 was used to facilitate such relationship analysis. No information is available on how the approach was evaluated.

4.7. 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 of 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.

4.7.1. Evolutionary couplings and change predictions

Zimmermann *et al.* [15] aimed to identify co-occurring changes in a software system. The purpose was 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 [15]. The subsequent entity changes in the *CVS* repository are grouped as a transaction using a sliding-window technique [15]. An association-rule mining technique was 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 was reported in [33] that allows prediction of additions to and deletion from entities. The tool *ROSE* was 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 versus coarse), maintenance (modified only), multidimensional (addition and deletion), history, and recent changes. Eight open-source projects were 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) was introduced to assess the ‘interactive power’ of the *ROSE* tool.

The average precision, recall, and feedback values taken across the given eight projects are as follows.

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

The average feedback values in the 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 the case of projects undergoing rapid renames and moves.

A similar approach was taken by Ying *et al.* [34] for source code change prediction at a file level. An association-mining technique based on FP-tree itemset mining was used. The mined rules were classified into the categories ‘surprising’, ‘neutral’, or ‘obvious’ to indicate their level of interest. The technique was evaluated on the version histories of *Mozilla* and *Eclipse* projects.

4.7.2. Mining usage patterns

Livshits and Zimmermann [35] presented 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 presented a corrective ranking (i.e., based on past changes that fixed bugs) to order the mined patterns. The objective of this work was to determine useful usage patterns and their violations.

The hypothesis was that violations of useful patterns are potential sources of errors. The patterns were classified into valid patterns, likely error patterns, and unlikely patterns. A snapshot of the source code was instrumented to obtain the run-time information of method calls. A candidate pattern mined from the version archive was 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 was considered as an error pattern. The approach was 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 has also been used for the problem of uncovering frequent patterns of co-changes.

4.7.3. Ordered change patterns

Kagdi *et al.* [36] 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 defined six heuristics for grouping the 'related' change-sets in a source code repository. Given such groups, sequences of files that frequently change together were uncovered using sequential-pattern mining. For example, sequences of changed-files such as $\{f1\} \rightarrow \{f2\}$ and $\{f4\} \rightarrow \{f5\}$ were uncovered. The sequence $\{f1\} \rightarrow \{f2\}$ indicates that the changes in $\{f1\}$ *happen 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 was demonstrated on a subset of the *KDE* source code repository. In other works, Burch *et al.* [82] presented a tool that supports visualization of association rules and sequence rules, El-Ramly and Stroulia [95] used sequence mining to detect patterns of user activities from the system-user interaction data, and Xie and Pei [94] 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.

4.8. Information-retrieval 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 e-mails makes IR an attractive choice. In this section, we discuss IR techniques applied to MSR.

4.8.1. 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 was presented by Mockus and Votta [77]. An automatic keyword clustering and classification (heuristic-based) algorithm was 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 three types (purposes or reasons) of 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 relate 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% of the MRs were later inferred to be corrective following manual validation by the authors. The percentage of the total number of deltas, lines added, lines deleted, and lines left unmodified are as follows.

- 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%.
- 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 of 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 for 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 indicated that corrective changes were perceived to be most hard, followed by perfective changes, and the inspection changes were perceived to be 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.

4.8.2. Change prediction

Canfora and Cerulo [56] 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 IR method was used to index the changed files in the *CVS* repositories with the textual description of past BRs in the *Bugzilla* repository and the *CVS* commit messages. A BR 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 BR. The query is formed from the textual description of a BR.

The approach was evaluated on four open-source projects *Kcalc*, *Kpdf*, *Kspread*, and *Firefox*. Precision and recall metrics were used as the assessment metrics. A validation technique known as leave-out-one was 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 were 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 BRs. 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 were reported for 30 queries on each project. The precisions of *Kcalc*, *Kpdf*, *Kspread*, and *Firefox* were reported in the range [38%, 78%], in the range [36%, 45%], 39%, and 36%, respectively. The recall of *Kcalc*, *Kpdf*, *Kspread*, and *Firefox* were reported in the range [82%, 98%], in the range [70%, 85%], 79%, and 67%, respectively.

A further extension of this work was reported in [57] where the prediction was 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 with file-level granularity. However, the cost of indexing at line-level is of the order of hours compared with the order of seconds with file-level granularity.

4.8.3. 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 was argued by Hayes *et al.* [59]. 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 suggested that only the refined results obtained by an analyst (and not those directly produced by a data-mining tool) should be made available to others (i.e., tools/human).

Two case studies on the *MODIS* dataset were described in [59] 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 was 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 IR technique was 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 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 a relatively

¶Here, the term 'data mining' is used in a broader context, including IR methods.

longer time to complete and produced no worse final (refined) results than data sets with high recall. Overall, no performance-improving pattern was discovered. While this work was 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.

4.8.4. *New developer assistance*

Cubranic *et al.* [60,61] described a tool, *Hipikat*, to assist new developers (not necessarily novice) on a project. Various artifacts (e.g., source code, e-mail, and BRs) produced in the project were integrated to form a *project memory*. A vector-based IR method was used to draw the similarity between artifacts. Other relationships between artifacts were formed by using heuristics, e.g., MRs in *Bugzilla* are related to the files in *CVS* by matching bug-ids 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 [61] 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 zero 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 reader to [61]. 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 [86] 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 e-mail addresses and URLs found in unstructured text.

4.8.5. *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 evolve 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, was examined by Antoniol *et al.* [62]. 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 of the evolving software modules are transformed into 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 was applied to the *Linux* kernel versions 1.1.0 to 1.3.1.0 consisting of 1788 files and 211 releases. The size of the modules was defined by the 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 a decreasing number of similar pairs. Similarly, increasing the threshold requirement (e.g., 10^{-3} to 10^{-5}) resulted in a 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.

4.9. 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 are 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 allow machine-learning techniques to be applied for discovering and forming classification and prediction models within the context of MSR.

4.9.1. Maintenance relevance relations

A classification-learning technique is used by Shirabad *et al.* [37–39] 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 non-relevant relation between a pair of files to form the learning and testing sets. A fixed time period between time T_1 and T_2 ($T_2 < T_1$) 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 T_3 and T_2 is chosen ($T_3 < T_1$) and all of the relations between a pair of files that are not marked as relevant are considered non-relevant. The classifier takes as input a pair of files and assigns the co-update relation between them to either relevant or non-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 PRs, program traces, memory dumps, file comments). Note that the text-based attributes are represented by a Boolean bag of words (all of 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 4700 files and 1.9 MLOC written in a high-level programming language and assembly language [39]. 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 PR attributes generate better classifiers than those of syntactic attributes. The comment attributes generated classifiers do not perform on a par with those generated with the PR 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.

4.9.2. Triage bug reports

Anvik *et al.* [63] used a supervised learning (i.e., support vector machine algorithm) in order to recommend a list of potential developers for resolving a BR. Past reports in the *Bugzilla* repository are used to produce a classifier. The authors developed project-specific heuristics to train the classifier instead of directly using the *assigned-to* field of a BR. This was done to avoid incorrect assignment of BRs 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 BR 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.

4.10. Social network analysis

Social network analysis [104] is a technique widely used in social and behavioral sciences for deriving and measuring ‘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.

4.10.1. Developers roles and contributions

An approach based on social network analysis to group developers using the logs (deltas) stored in the *CVS* repository was proposed by Huang and Liu [64]. The log data were analyzed to determine developers’ contributions at a module (directory) level. This information was 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 was 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 lower the distance-centrality value of node, the greater the connection with (possibly many) other nodes. The authors report their findings on six projects selected from *SourceForge*. In one project, all of 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.* [105] 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 were suggested and interpreted. A case study and the results were discussed on *Apache*, *GNOME*, and *KDE* systems.

4.10.2. Inter-projects collaboration

A visualization tool, *Graphmania*, with the goal of supporting cross-project knowledge sharing and collaboration was presented by Ohira *et al.* [65]. The authors observed 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 network: 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 of the nodes and edges of the developer networks and project networks. In addition, 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 was described. Only small subsets (sub-graphs) of the three networks were presented. Further analysis showed 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).

5. DISCUSSION AND OPEN ISSUES

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.

5.1. 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.* [33] 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 [33], 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) need 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 is used to find logical/evolutionary couplings between source code entities. These couplings are termed ‘hidden’ dependencies as they are solely based on the 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.* [106] analyzed change-sets from a *CVS* repository to distinguish between 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 the version history with this information (i.e., reduce false positives). Their study on an *Eclipse* plugin found over 31% of change-sets with no structural changes and over 51% of change-sets with at least one non-structural change. In one of the rare cases, Ying *et al.* [34] 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.

5.2. Historical context: how many versions?

Software repositories bring a rich history of software development and evolution. One goal of MSR is to undercover 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 1000 versions. The information about the changes in the first 50 versions may be totally irrelevant for predicting the changes in version 1001. 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 few 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). The minimum requirement would be the past versions beginning from when the refactoring started 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.

5.3. 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 of these factors are subject to threats to validity.

Gasser *et al.* [16] identified 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 suggested a research infrastructure to deal with such challenges and to serve as a benchmark to facilitate comparative and collaborative research. They discussed 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 *et al.* further suggested 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 [10,18].

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.

6. CONCLUDING REMARKS

Over 80 investigations were surveyed that examined 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, BRs, 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 MSR 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.

ACKNOWLEDGEMENTS

We thank the reviewers for their detailed comments and suggestions. These were instrumental in helping us improve the presentation of the survey.

REFERENCES

1. Lehman M. On understanding laws, evolution and conservation in the large program life cycle. *Journal of Systems and Software* 1980; **1**(3):213–221.
2. Lehman M, Perry D, Ramil JF. On evidence supporting the FEAST hypothesis and the laws of software evolution. *Proceedings 5th International Symposium on Software Metrics (METRICS'98)*. IEEE Computer Society Press: Los Alamitos CA, 1998; 84–88.
3. Lehman M, Ramil JF. An approach to a theory of software evolution. *Proceedings International Workshop on Principles of Software Evolution (IWPSE'01)*. IEEE Computer Society Press: Los Alamitos CA, 2001; 70–74.
4. Lehman M, Ramil JF. Evolution in software and related areas. *Proceedings International Workshop on Principles of Software Evolution (IWPSE'01)*. IEEE Computer Society Press: Los Alamitos CA, 2001; 1–16.
5. Lehman MM, Belady LA. *Program Evolution: Processes of Software Change*. Academic Press: New York NY, 1985.
6. Ramil JF, Lehman M. Metrics of software evolution as effort predictors—a case study. *Proceedings 16th IEEE International Conference on Software Maintenance (ICSM'00)*. IEEE Computer Society Press: Los Alamitos CA, 2000; 163–172.
7. Weiss DM, Basili VR. Evaluating software development by analysis of changes: Some data from the software engineering laboratory. *IEEE Transactions on Software Engineering* 1985; **11**(2):157–168.
8. Eick SG, Graves TL, Karr AF, Marron JS, Mockus A. Does code decay? Assessing the evidence from change management data. *IEEE Transactions on Software Engineering* 2001; **27**(1):1–12.
9. Kagdi H, Collard ML, Maletic JI. Towards a taxonomy of approaches for mining of source code repositories. *Proceedings 2nd International Workshop on Mining Software Repositories (MSR'05)*. ACM Press: New York NY, 2005; 90–94.
10. German DM, Cubranic D, Storey MAD. A framework for describing and understanding mining tools in software development. *Proceedings 2nd International Workshop on Mining Software Repositories (MSR'05)*. ACM Press: New York NY, 2005; 95–99.

11. Kim M, Notkin D. Program element matching for multi-version program analyses. *Proceedings 3rd International Workshop on Mining Software Repositories (MSR'06)*. ACM Press: New York NY, 2006; 58–64.
12. Buckley J, Mens T, Zenger M, Rashid A, Kniesel G. Towards a taxonomy of software change. *Journal of Software Maintenance and Evolution: Research and Practice* 2005; **17**(5):309–332. DOI:10.1002/smr.319.
13. Robles G, González-Barahona JM, Ghosh RA. GlueTheos: Automating the retrieval and analysis of data from publicly available software repositories. *Proceedings 1st International Workshop on Mining Software Repositories (MSR'04)*. University of Waterloo: Waterloo ON, 2004; 28–31. Available at: http://plg.uwaterloo.ca/∼aeehassa/home/pubs/MSR2004ProceedingsFINAL_IEE_Acrobat4.pdf [1 February 2007].
14. Alonso O, Devanbu PT, Gertz M. Database techniques for the analysis and exploration of software repositories. *Proceedings 1st International Workshop on Mining Software Repositories (MSR'04)*. University of Waterloo: Waterloo ON, 2004; 37–41. Available at: http://plg.uwaterloo.ca/∼aeehassa/home/pubs/MSR2004ProceedingsFINAL_IEE_Acrobat4.pdf [1 February 2007].
15. Zimmermann T, Weißgerber P, Diehl S, Zeller A. Mining version histories to guide software changes. *Proceedings 26th International Conference on Software Engineering (ICSE'04)*. IEEE Computer Society Press: Los Alamitos CA, 2004; 563–572.
16. Gasser L, Ripoche G, Sandusky RJ. Research infrastructure for empirical science of F/OSS. *Proceedings 1st International Workshop on Mining Software Repositories (MSR'04)*. University of Waterloo: Waterloo ON, 2004; 12–16. Available at: http://plg.uwaterloo.ca/∼aeehassa/home/pubs/MSR2004ProceedingsFINAL_IEE_Acrobat4.pdf [1 February 2007].
17. Conklin M, Howison J, Crowston K. Collaboration using OSSmole: A repository of FLOSS data and analyses. *Proceedings 2nd International Workshop on Mining Software Repositories (MSR'05)*. ACM Press: New York NY, 2005; 116–120.
18. German DM. Mining CVS repositories, the SoftChange experience. *Proceedings 1st International Workshop on Mining Software Repositories (MSR'04)*. University of Waterloo: Waterloo ON, 2004; 17–21. Available at: http://plg.uwaterloo.ca/∼aeehassa/home/pubs/MSR2004ProceedingsFINAL_IEE_Acrobat4.pdf [1 February 2007].
19. Howison J. Crowston K. The perils and pitfalls of mining Sourceforge. *Proceedings 1st International Workshop on Mining Software Repositories (MSR'04)*. University of Waterloo: Waterloo ON, 2004; 7–11. Available at: http://plg.uwaterloo.ca/∼aeehassa/home/pubs/MSR2004ProceedingsFINAL_IEE_Acrobat4.pdf [1 February 2007].
20. Gall H, Hajek K, Jazayeri M. Detection of logical coupling based on product release history. *Proceedings 14th IEEE International Conference on Software Maintenance (ICSM'98)*. IEEE Computer Society Press: Los Alamitos CA, 1998; 190–199.
21. German DM. An empirical study of fine-grained software modifications. *Proceedings 20th IEEE International Conference on Software Maintenance (ICSM'04)*. IEEE Computer Society Press: Los Alamitos CA, 2004; 316–325.
22. Chen A, Chou E, Wong J, Yao AY, Zhang Q, Zhang S, Michail A. CVSSearch: Searching through source code using CVS comments. *Proceedings 17th IEEE International Conference on Software Maintenance (ICSM'01)*. IEEE Computer Society Press: Los Alamitos CA, 2001; 364–373.
23. Fischer M, Pinzger M, Gall H. Populating a release history database from version control and bug tracking systems. *Proceedings 19th IEEE International Conference on Software Maintenance (ICSM'03)*. IEEE Computer Society Press: Los Alamitos CA, 2003; 23–32.
24. Zou L, Godfrey MW. Detecting merging and splitting using origin analysis. *Proceedings 10th Working Conference on Reverse Engineering (WCRE'03)*. IEEE Computer Society Press: Los Alamitos CA, 2003; 146–154.
25. Askari M, Holt R. Information theoretic evaluation of change prediction models for large-scale software. *Proceedings 3rd International Workshop on Mining Software Repositories (MSR'06)*. ACM Press: New York NY, 2006; 126–132.
26. Gall H, Jazayeri M, Krajewski J. CVS release history data for detecting logical couplings. *Proceedings 6th International Workshop on Principles of Software Evolution (IWPSE'03)*. IEEE Computer Society Press: Los Alamitos CA, 2003; 13–23.
27. Fischer M, Oberleitner J, Ratzinger J, Gall H. Mining evolution data of a product family. *Proceedings 2nd International Workshop on Mining Software Repositories (MSR'05)*. ACM Press: New York NY, 2005; 12–16.
28. Sliwerski J, Zimmermann T, Zeller A. When do changes induce fixes? *Proceedings 2nd International Workshop on Mining Software Repositories (MSR'05)*. ACM Press: New York NY, 2005; 24–28.
29. Hindle A, German DM. SCQL: A formal model and a query language for source control repositories. *Proceedings 2nd International Workshop on Mining Software Repositories (MSR'05)*. ACM Press: New York NY, 2005; 100–104.
30. Kim S, Whitehead EJ, Bevan J. Analysis of signature change patterns. *Proceedings 2nd International Workshop on Mining Software Repositories (MSR'05)*. ACM Press: New York NY, 2005; 64–68.
31. Maletic JI, Collard ML. Supporting source code difference analysis. *Proceedings 20th IEEE International Conference on Software Maintenance (ICSM'04)*. IEEE Computer Society Press: Los Alamitos CA, 2004; 210–219.

32. Van Rysselberghe F, Demeyer S. Mining version control systems for FACs (frequently applied changes). *Proceedings 1st International Workshop on Mining Software Repositories (MSR'04)*. IEE: Stevenage, 2004; 48–52.
33. Zimmermann T, Zeller A, Weißgerber P, Diehl S. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering* 2005; **31**(6):429–445.
34. Ying ATT, Murphy GC, Ng R, Chu-Carroll MC. Predicting source code changes by mining change history. *IEEE Transactions on Software Engineering* 2004; **30**(9):574–586.
35. Livshits B, Zimmermann T. DynaMine: Finding common error patterns by mining software revision histories. *Proceedings 13th International Symposium on Foundations of Software Engineering (ESEC/FSE'05)*. ACM Press: New York NY, 2005; 296–305.
36. Kagdi H, Yusuf S, Maletic JI. Mining sequences of changed-files from version histories. *Proceedings 3rd International Workshop on Mining Software Repositories (MSR'06)*. ACM Press: New York NY, 2006; 47–53.
37. Shirabad JS, Lethbridge TC, Matwin S. Supporting software maintenance by mining software update records. *Proceedings 17th IEEE International Conference on Software Maintenance (ICSM'01)*. IEEE Computer Society Press: Los Alamitos CA, 2001; 22–31.
38. Shirabad JS, Lethbridge TC, Matwin S. Mining the maintenance history of a legacy software system. *Proceedings 19th IEEE International Conference on Software Maintenance (ICSM'03)*. IEEE Computer Society Press: Los Alamitos CA, 2003; 95–104.
39. Shirabad JS, Lethbridge TC, Matwin S. Mining the software change repository of a legacy telephony system. *Proceedings 1st International Workshop on Mining Software Repositories (MSR'04)*. University of Waterloo: Waterloo ON, 2004; 53–57. Available at: http://plg.uwaterloo.ca/∼aeehassa/home/pubs/MSR2004ProceedingsFINAL_IEE_Acrobat4.pdf [1 February 2007].
40. Dinh-Trong TT, Bieman JM. The FreeBSD project: A replication case study of open source development. *IEEE Transactions on Software Engineering* 2005; **31**(6):481–494.
41. Robles G, González-Barahona JM. Developer identification methods for integrated data from various sources. *Proceedings 2nd International Workshop on Mining Software Repositories (MSR'05)*. ACM Press: New York NY, 2005; 106–110.
42. Sandusky RJ, Gasser L, Ripoche G. Bug report networks: Varieties, strategies, and impacts in a F/OSS development community. *Proceedings 1st International Workshop on Mining Software Repositories (MSR'04)*. University of Waterloo: Waterloo ON, 2004; 80–84. Available at: http://plg.uwaterloo.ca/∼aeehassa/home/pubs/MSR2004ProceedingsFINAL_IEE_Acrobat4.pdf [1 February 2007].
43. Ostrand TJ, Weyuker EJ. A tool for mining defect-tracking systems to predict fault-prone files. *Proceedings 1st International Workshop on Mining Software Repositories (MSR'04)*. University of Waterloo: Waterloo ON, 2004; 85–89. Available at: http://plg.uwaterloo.ca/∼aeehassa/home/pubs/MSR2004ProceedingsFINAL_IEE_Acrobat4.pdf [1 February 2007].
44. Robles G, González-Barahona JM, Michlmayr M, Amor JJ. Mining large software compilations over time: Another perspective of software evolution. *Proceedings 3rd International Workshop on Mining Software Repositories (MSR'06)*. ACM Press: New York NY, 2006; 3–9.
45. Williams CC, Hollingsworth JK. Bug driven bug finders. *Proceedings 1st International Workshop on Mining Software Repositories (MSR'04)*. University of Waterloo: Waterloo ON, 2004; 70–74. Available at: http://plg.uwaterloo.ca/∼aeehassa/home/pubs/MSR2004ProceedingsFINAL_IEE_Acrobat4.pdf [1 February 2007].
46. Williams CC, Hollingsworth JK. Automatic mining of source code repositories to improve bug finding techniques. *IEEE Transactions on Software Engineering* 2005; **31**(6):466–480.
47. Selby RW. Enabling reuse-based software development of large-scale systems. *IEEE Transactions on Software Engineering* 2005; **31**(6):495–510.
48. Williams CC, Hollingsworth JK. Recovering system specific rules from software repositories. *Proceedings 2nd International Workshop on Mining Software Repositories (MSR'05)*. ACM Press: New York NY, 2005; 7–11.
49. Görg C, Weißgerber P. Error detection by refactoring reconstruction. *Proceedings 2nd International Workshop on Mining software repositories (MSR'05)*. ACM Press: New York NY, 2005; 29–33.
50. Ying ATT, Wright JL, Abrams S. Source code that talks: An exploration of Eclipse task comments and their implication to repository mining. *Proceedings 2nd International Workshop on Mining software repositories (MSR'05)*. ACM Press: New York NY, 2005; 53–57.
51. Capiluppi A, Morisio M, Ramil JF. Structural evolution of an open source system: A case study. *Proceedings 12th IEEE International Workshop on Program Comprehension (IWPC'04)*. IEEE Computer Society Press: Los Alamitos CA, 2004; 172–182.
52. Menzies T, Di Stefano JS, Cunanan C, Chapman R. Mining repositories to assist in project planning and resource allocation. *Proceedings 1st International Workshop on Mining Software Repositories (MSR'04)*. University of Waterloo: Waterloo ON, 2004; 75–79. Available at: http://plg.uwaterloo.ca/∼aeehassa/home/pubs/MSR2004ProceedingsFINAL_IEE_Acrobat4.pdf [1 February 2007].

53. Van Rysselberghe F, Demeyer S. Studying software evolution information by visualizing the change history. *Proceedings 20th IEEE International Conference on Software Maintenance (ICSM'04)*. IEEE Computer Society Press: Los Alamitos CA, 2004; 328–37.
54. Ratzinger J, Fischer M, Gall H. Improving evolvability through refactoring. *Proceedings 2nd International Workshop on Mining software repositories (MSR'05)*. ACM Press: New York NY, 2005; 69–73.
55. Kim M, Notkin D. Using a clone genealogy extractor for understanding and supporting evolution of code clones. *Proceedings 2nd International Workshop on Mining Software Repositories (MSR'05)*. ACM Press: New York NY, 2005; 17–21.
56. Canfora G, Cerulo L. Impact analysis by mining software and change request repositories. *Proceedings 11th IEEE International Symposium on Software Metrics (METRICS'05)*. IEEE Computer Society Press: Los Alamitos CA, 2005; 29–37.
57. Canfora G, Cerulo L. Fine grained indexing of software repositories to support impact analysis. *Proceedings 3rd International Workshop on Mining Software Repositories (MSR'06)*. ACM Press: New York NY, 2006; 105–111.
58. Ohba M, Gondow K. Toward mining concept keywords from identifiers in large software projects. *Proceedings 2nd International Workshop on Mining Software Repositories (MSR'05)*. ACM Press: New York NY, 2005; 48–52.
59. Hayes JH, Dekhtyar A, Sundaram S. Text mining for software engineering: How analyst feedback impacts final results. *Proceedings 2nd International Workshop on Mining Software Repositories (MSR'05)*. ACM Press: New York NY, 2005; 58–62.
60. Cubranic D, Murphy GC. Hipikat: Recommending pertinent software development artifacts. *Proceedings 25th International Conference on Software Engineering (ICSE'03)*. IEEE Computer Society Press: Los Alamitos CA, 2003; 408–418.
61. Cubranic D, Murphy GC, Singer J, Booth KS. Hipikat: A project memory for software development. *IEEE Transactions on Software Engineering* 2005; **31**(6):446–465.
62. Antoniol G, Rollo VF, Venturi G. Linear predictive coding and Cepstrum coefficients for mining time variant information from software repositories. *Proceedings 2nd International Workshop on Mining Software Repositories (MSR'05)*. ACM Press: New York NY, 2005; 74–78.
63. Anvik J, Hiew L, Murphy GC. Who should fix this bug? *Proceedings 28th International Conference on Software Engineering (ICSE'06)*. ACM Press: New York NY, 2006; 361–370.
64. Huang S-K, Liu K-M. Mining version histories to verify the learning process of legitimate peripheral participants. *Proceedings 2nd International Workshop on Mining Software Repositories (MSR'05)*. ACM Press: New York NY, 2005; 84–78.
65. Ohira M, Ohsugi N, Ohoka T, Matsumoto K-I. Accelerating cross-project knowledge collaboration using collaborative filtering and social networks. *Proceedings 2nd International Workshop on Mining Software Repositories (MSR'05)*. ACM Press: New York NY, 2005; 111–115.
66. Ohira M, Yokomori R, Sakai M, Matsumoto K, Inoue K, Torii K. Empirical project monitor: A tool for mining multiple project data. *Proceedings 1st International Workshop on Mining Software Repositories (MSR'04)*. University of Waterloo: Waterloo ON, 2004; 42–46. Available at: http://plg.uwaterloo.ca/~aeehassa/home/pubs/MSR2004ProceedingsFINAL_IEE_Acrobat4.pdf [1 February 2007].
67. Purushothaman R, Perry DE. Towards understanding the rhetoric of small changes. *Proceedings 1st International Workshop on Mining Software Repositories (MSR'04)*. University of Waterloo: Waterloo ON, 2004; 90–94. Available at: http://plg.uwaterloo.ca/~aeehassa/home/pubs/MSR2004ProceedingsFINAL_IEE_Acrobat4.pdf [1 February 2007].
68. Purushothaman R, Perry DE. Toward understanding the rhetoric of small source code changes. *IEEE Transactions on Software Engineering* 2005; **31**(6):511–526.
69. Hassan AE, Holt RC. Predicting change propagation in software systems. *Proceedings 20th IEEE International Conference on Software Maintenance (ICSM'04)*. IEEE Computer Society Press: Los Alamitos CA, 2004; 284–293.
70. Raghavan S, Rohana R, Leon D, Podgurski A, Augustine V. Dex: A semantic-graph differencing tool for studying changes in large code bases. *Proceedings 20th IEEE International Conference on Software Maintenance (ICSM'04)*. IEEE Computer Society Press: Los Alamitos CA, 2004; 188–197.
71. Neamtiu I, Foster JS, Hicks M. Understanding source code evolution using abstract syntax tree matching. *Proceedings 2nd International Workshop on Mining Software Repositories (MSR'05)*. ACM Press: New York NY, 2005; 2–6.
72. Godfrey MW, Zou L. Using origin analysis to detect merging and splitting of source code entities. *IEEE Transactions on Software Engineering* 2005; **31**(2):166–181.
73. Nikora AP, Munson JC. Understanding the nature of software evolution. *Proceedings 19th IEEE International Conference on Software Maintenance (ICSM'03)*. IEEE Computer Society Press: Los Alamitos CA, 2003; 83–93.
74. Tu Q, Godfrey MW. An integrated approach for studying architectural evolution. *Proceedings 10th International Workshop on Program Comprehension (IWPC'02)*. IEEE Computer Society Press: Los Alamitos CA, 2002; 127–136.

75. Holt R, Pak JY. GASE: Visualizing software evolution-in-the-large. *Proceedings 3rd Working Conference on Reverse Engineering (WCRE'96)*. IEEE Computer Society Press: Los Alamitos CA, 1996; 163–167.
76. Gall H, Jazayeri M, Riva C. Visualizing software release histories: The use of color and third dimension. *Proceedings 15th IEEE International Conference on Software Maintenance (ICSM'99)*. IEEE Computer Society Press: Los Alamitos CA, 1999; 99–108.
77. Mockus A, Votta LG. Identifying reasons for software changes using historic databases. *Proceedings 16th IEEE International Conference on Software Maintenance (ICSM'00)*. IEEE Computer Society Press: Los Alamitos CA, 2000; 120–130.
78. Görg C, Weißgerber P. Detecting and visualizing refactorings from software archives. *Proceedings 13th International Workshop on Program Comprehension (IWPC'05)*. IEEE Computer Society Press: Los Alamitos CA, 2005; 205–214.
79. Bieman JM, Andrews AA, Yang HJ. Understanding change-proneness in OO software through visualization. *Proceedings 11th IEEE International Workshop on Program Comprehension (IWPC'03)*. IEEE Computer Society Press: Los Alamitos CA, 2003; 44–53.
80. German DM. Using software trails to reconstruct the evolution of software. *Journal of Software Maintenance and Evolution: Research and Practice* 2004; **16**(6):367–384. DOI: 10.1002/smr.301.
81. Beyer D, Noack A. Clustering software artifacts based on frequent common changes. *Proceedings 13th International Workshop on Program Comprehension (IWPC'05)*. IEEE Computer Society Press: Los Alamitos CA, 2005; 259–268.
82. Burch M, Diehl S, Weißgerber P. Visual data mining in software archives. *Proceedings ACM Symposium on Software Visualization (SoftVis'05)*. ACM Press: New York NY, 2005; 37–46.
83. Chen K, Schach SR, Yu L, Offutt J, Heller GZ. Open-source change logs. *Empirical Software Engineering* 2004; **9**(3):197–210.
84. Kim S, Pan K, Whitehead EJ Jr. Micro pattern evolution. *Proceedings 3rd International Workshop on Mining Software Repositories (MSR'06)*. ACM Press: New York NY, 2006; 40–46.
85. Riva C. Visualizing software release histories with 3DSoftVis. *Proceedings 22nd International Conference on Software Engineering (ICSE'00)*. ACM Press: New York NY, 2000; 789.
86. Venolia G. Textual allusions to artifacts in software-related repositories. *Proceedings 3rd International Workshop on Mining Software Repositories (MSR'06)*. ACM Press: New York NY, 2006; 151–154.
87. Nagappan N, Ball T, Zeller A. Mining metrics to predict component failures. *Proceedings 28th International Conference on Software Engineering (ICSE'06)*. ACM Press: New York NY, 2006; 452–461.
88. Sager T, Bernstein A, Pinzger M, Kiefer C. Detecting similar java classes using tree algorithms. *Proceedings 3rd International Workshop on Mining Software Repositories (MSR'06)*. ACM Press: New York NY, 2006; 66–71.
89. Dig D, Comertoglu C, Marinov D, Johnson R. Automated detection of refactorings in evolving components. *Proceedings European Conference on Object-Oriented Programming (ECOOP'06)*. Springer: Berlin, 2006; 404–428.
90. Dig D, Johnson R. How do APIs evolve? A story of refactoring. *Journal of Software Maintenance and Evolution: Research and Practice* 2006; **18**(2):83–107. DOI: 10.1002/smr.328.
91. Godfrey M, Dong X, Kapser C, Zou L. Four interesting ways in which history can teach us about software. *Proceedings 1st International Workshop on Mining Software Repositories (MSR'04)*. University of Waterloo: Waterloo ON, 2004; 58–62. Available at:
http://plg.uwaterloo.ca/∼aeehassa/home/pubs/MSR2004ProceedingsFINAL_IEE_Acrobat4.pdf [1 February 2007].
92. Henkel J, Diwan A. CatchUp!: Capturing and replaying refactorings to support API evolution. *Proceedings 27th International Conference on Software Engineering (ICSE'05)*. ACM Press: New York NY, 2005; 274–283.
93. Weißgerber P, Diehl S. Are refactorings less error-prone than other changes? *Proceedings 3rd International Workshop on Mining Software Repositories (MSR'06)*. ACM Press: New York NY, 2006; 112–118.
94. Xie T, Pei J. MAPO: Mining API usages from open source repositories. *Proceedings 3rd International Workshop on Mining Software Repositories (MSR'06)*. ACM Press: New York NY, 2006; 54–57.
95. El-Ramly M, Stroulia E. Mining software usage data. *Proceedings 1st International Workshop on Mining Software Repositories (MSR'04)*. University of Waterloo: Waterloo ON, 2004; 64–68. Available at:
http://plg.uwaterloo.ca/∼aeehassa/home/pubs/MSR2004ProceedingsFINAL_IEE_Acrobat4.pdf [1 February 2007].
96. Mockus A, Fielding T, Herbsleb D. Two case studies of open source software development: Apache and Mozilla. *ACM Transactions on Software Engineering and Methodology* 2002; **11**(3):309–346.
97. Koch S, Schneider G. Effort, cooperation and coordination in an open source software project: Gnome. *Information Systems Journal* 2002; **12**(1):27–42.
98. Robles G, Koch S, González-Barahona JM. Remote analysis and measurement of Libre software systems by means of the CVSAnalY tool. *Proceedings 2nd Remote Analysis of Software Systems (RAMSS'04)*. IEE: Stevenage, 2004; 51–55.
99. Gil JY, Maman I. Micro patterns in Java code. *Proceedings 20th Object Oriented Programming Systems Languages and Applications (OOPSLA 05)*. ACM Press: New York NY, 2005; 97–116.

100. Demeyer S, Tichelaar S, Steyaert P. FAMIX 2.0: The FAMOOS information exchange model. http://www.iam.unibe.ch/∼famoos/FAMIX/ [20 June 2006].
101. Maletic JI, Marcus A, Collard ML. A task oriented view of software visualization. *Proceedings 1st IEEE Workshop of Visualizing Software for Understanding and Analysis (VISSOFT'02)*. IEEE Computer Society Press: Los Alamitos CA, 2002; 32–40.
102. Fowler M. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley: Reading MA, 1999.
103. Kim M, Bergman L, Lau T, Notkin D. An ethnographic study of copy and paste programming practices in OOPL. *Proceedings International Symposium on Empirical Software Engineering (ISESE'04)*. IEEE Computer Society Press: Los Alamitos CA, 2004; 83–92.
104. Wasserman S, Faust K. *Social Network Analysis: Methods and Applications*. Cambridge University Press: Cambridge, 1994.
105. Lopez-Fernandez L, Robles G, González-Barahona JM. Applying social network analysis to the information in CVS repositories. *Proceedings 1st International Workshop on Mining Software Repositories (MSR'04)*. University of Waterloo: Waterloo ON, 2004; 101–105. Available at: http://plg.uwaterloo.ca/∼aeehassa/home/pubs/MSR2004ProceedingsFINAL_IEE_Acrobat4.pdf [1 February 2007].
106. Fluri B, Gall H, Pinzger M. Fine-grained analysis of change couplings. *Proceedings 5th International Workshop on Source Code Analysis and Manipulation (SCAM'05)*. IEEE Computer Society Press: Washington DC, 2005; 66–74.

AUTHORS' BIOGRAPHIES

Huzefa Kagdi is a Doctoral Candidate in the Department of Computer Science at Kent State University in Ohio, U.S.A. His research interests are in mining software repositories, source code representations and analysis, and UML visualization for supporting evolution of large-scale software systems. He received his MS in Computer Science from Kent State University, U.S.A., and BE in Computer Engineering from Birla Vishwakarma Mahavidyalaya, India.

Michael L. Collard is a Visiting Assistant Professor in the Department of Mathematics and Computer Science at Ashland University in Ohio, U.S.A. His research interests are in source code and source model representation, source code analysis, transformation/refactoring, and differencing for software evolution. He received his PhD, MS, and BS in Computer Science from Kent State University.

Jonathan I. Maletic is an Associate Professor in the Department of Computer Science at Kent State University in Ohio, U.S.A. His research interests are centered on software evolution and he has authored over 60 refereed publications in the areas of analysis, transformation, comprehension, traceability, and visualization of software. He received his PhD and MS in Computer Science from Wayne State University and BS in Computer Science from The University of Michigan–Flint.