The Road Ahead for Mining Software Repositories

Ahmed E. Hassan
Software Analysis and Intelligence Lab (SAIL)
School of Computing, Queen's University, Canada
ahmed@cs.queensu.ca

Abstract

Source control repositories, bug repositories, archived communications, deployment logs, and code repositories are examples of software repositories that are commonly available for most software projects. The Mining Software Repositories (MSR) field analyzes and cross-links the rich data available in these repositories to uncover interesting and actionable information about software systems. By transforming these repositories from static record-keeping ones into active repositories, we can guide decision processes in modern software projects. For example, data in source control repositories, traditionally used to archive code, could be linked with data in bug repositories to help practitioners propagate complex changes and to warn them about risky code based on prior changes and bugs. In this paper, we present a brief history of the MSR field and discuss several recent achievements and results of using MSR techniques to support software research and practice. We then discuss the various opportunities and challenges that lie in the road ahead for this important and emerging field.

1 Introduction

Prior experiences and dominant patterns are the driving force for many decision-processes in modern software organizations. Practitioners often rely on their experience, intuition and gut feeling in making important decisions. Managers allocate development and testing resources based on their experience in previous projects and their intuition about the complexity of the new project relative to prior projects. Developers commonly use their experience when adding a new feature or fixing a bug. Testers usually prioritize the testing of features that are known to be error prone based on field and bug reports. Software repositories contain a wealth of valuable information about software projects. Using the information stored in these repositories, practitioners can depend less on their intuition and experience, and depend more on historical and field data.

The Mining Software Repositories (MSR) field analyzes and cross-links the rich data available in software repositories to uncover interesting and actionable information about software systems and projects. Examples of software repositories are:

Historical repositories such as source control repositories, bug repositories, and archived communications record several information about the evolution and progress of a project.

Run-time repositories such as deployment logs contain information about the execution and the usage of an application at a single or multiple deployment sites.

Code repositories such as Sourceforge.net and Google code contain the source code of various applications developed by several developers.

Software repositories are commonly used in practice as record-keeping repositories and are rarely used to support decision making processes. For example, historical repositories are used to track the history of a bug or a feature, but are not commonly used to determine the expected resolution time of an open bug based on the resolution time of previously-closed bugs.

MSR researchers aim to transform these repositories from static record-keeping ones into active repositories which can guide decision processes in modern software projects. Mining these historical, run-time and code repositories, we can uncover useful and important patterns and information. Historical repositories capture important historical dependencies [20] between project artifacts, such as functions, documentation files, or configuration files. Developers can use this information to propagate changes to related artifacts, instead of only using static or dynamic code dependencies which may fail to capture important dependencies. For example, a change to the code which writes data to a file may require changes to the code which reads data from the file, although there exists no traditional dependencies (e.g., data and control flow) between both pieces of

978-1-4244-2655-3/08/$25.00 © 2008 IEEE

code. As for run-time repositories, they could be used to pinpoint execution anomaly by identifying dominant execution or usage patterns across deployments, and flagging deviations from these patterns. Code repositories could be used to identify dominant and correct library usage patterns by mining the usage of a library across many projects.

The MSR field is rapidly taking a central and important role in supporting software development practice and software engineering research. In this paper, we present a brief history of the MSR field and discuss several recent achievements and results of using MSR techniques to support software research and practice. We then discuss the various opportunities and challenges that lie in the road ahead for this important and emerging field.

2 A Brief History of the Mining Software Repositories (MSR) Field

The Mining Software Repositories (MSR) field is maturing thanks to the rich, extensive, and readily available software repositories. Table 1 lists several examples of software repositories which could be mined. Although these repositories are readily available for most large software projects, the data stored in these repositories has not been the focus of software engineering research until recently. This is owing primarily to the following two reasons:

Limited Access to Repositories. Companies in many cases are not willing to give researchers access to such detailed information about their software systems. Another possible source for software repositories is academic projects and systems. Unfortunately, the repositories of such projects and systems are not as rich nor as interesting as the those of long-lived widely-deployed commercial software systems. The earliest research work in the MSR field were based on the repositories of commercial software systems and were done in cooperation with a few commercial organizations such as NASA [7], AT&T [55, 14], Nortel [30], Nokia [20], Avaya [50], and Mitel [59].

Complexity of Data Extraction. Most repositories are not designed with automated bulk data-extraction and mining in mind, so they provide limited support for automated extraction. The complexity of automated data extraction hindered the adoption and integration of software repositories in other software engineering research. In many cases, software engineering researchers do not have the expertise required nor do they have the interest to extract data from software repositories. Extracting such data requires a great deal of effort and time from researchers, instead they are more interested in gaining convenient access to the extracted data in an easy to process format.

With the advent of open source systems, easy access to repositories of large software systems became a reality. Researchers now have access to rich repositories for large projects used by thousands of users and developed by hundreds of developers over extended periods of time. Early research in mining software repositories, e.g.[10], made use of open source repositories, thanks to the wide spread and growth of open source projects.

In an effort to bring together the practitioners and researchers working in this important and emerging field, the first International Workshop on Mining Software Repositories (MSR) was held at the International Conference on Software Engineering (ICSE), the flagship conference for Software Engineering. After four successful years as ICSE's largest workshop, MSR became a Working Conference in 2008. As a Working Conference, MSR recognizes the maturity and breadth of the work in the MSR field, while still encouraging free-form open discussions about the MSR field. An MSR challenge is also held on a yearly basis so researchers can compare their techniques towards a common problem or project. The MSR field continues to attract a large amount of interest within software engineering. A 2005 issue of the IEEE Transactions on Software Engineering (TSE) on the MSR topic received over 15% of all the submissions to the TSE in 2004 [27]. And MSR-related publications in top research venues continue to grow in size and quality with many MSR-related papers winning awards in top venues.

3 An Overview of MSR Achievements

MSR research focuses primarily on two aspects:

1. The creation of techniques to automate and improve the extraction of information from repositories.

2. The discovery and validation of novel techniques and approaches to mine important information from these repositories.

The first line of work is essential to ease the adoption of MSR techniques by others, and the second line of work demonstrates the importance and value of information stored in software repositories and encourages others to adopt MSR techniques.In this section we present a few interesting research results along a number of dimensions to demonstrate the benefits of using MSR techniques in solving several challenging and important software engineering problems. For a more extensive review of the work in the MSR field, we encourage the reader to refer to the "Mining Software Engineering Data Bibliography" maintained by Tao Xie at `http://ase.csc.ncsu.edu/dmse/` at North Carolina State University, and a survey paper by Kagdi *et al.* [32] on mining historical repositories.

Repository	Description
Source control repositories	These repositories record the development history of a project. They track all the changes to the source code along with meta-data about each change, e.g., the name of the developer who performed the change, the time the change was performed and a short message describing the change. Source control repositories are the most commonly available and used repository in software projects. CVS, subversion, Perforce, ClearCase are examples of source control repositories which are used in practice.
Bug repositories	These repositories track the resolution history of bug reports or feature requests that are reported by users and developers of large software projects. Bugzilla and Jira are examples of bug repositories.
Archived communications	These repositories track discussions about various aspects of a software project throughout its lifetime. Mailing lists, emails, IRC chats, and instant messages are examples of archived communications about a project.
Deployment logs	These repositories record information about the execution of a single deployment of a software application or different deployments of the same applications. For example, the deployment logs may record the error messages reported by an application at various deployment sites. The availability of deployment logs continues to increase at a rapid rate due to their use for remote issue resolution and due to recent legal acts. For instance, the Sarbanes-Oxley Act of 2002 [1] stipulates that the execution of telecommunication and financial applications must be logged.
Code repositories	These repositories archive the source code for a large number of projects. Sourceforge.net and Google code are examples of large code repositories.

Table 1. Examples of Software Repositories

3.1 Understanding Software Systems

Understanding large software systems remains a challenge for most software organizations. Documentations for large systems rarely exist and if they exist they are often not up-to-date. Moreover system experts are usually too busy to help novice developers, or may no longer be part of an organization. Information stored in historical software repositories, such as mailing lists and bug repositories, represent a group memory for a project. Such information is very valuable for current members of a project.

Cubranic *et al.* propose a tool called Hipikat which silently indexes historical repositories, and displays on-demand relevant historical information within the development environment [12]. While working on a particular change, Hipikat can display pertinent artifacts such as old emails and bug reports, discussing the code being viewed in the code editor.

Dependency graphs and the source code documentation offer a static view of a system and fail to reveal details about the history of a system or the rationale for its current state or design. For example, traditional dependency graphs cannot give the rationale behind an *Optimizer* function unexpectedly depending on a *Parser* function in a compiler. Such rationale is stored in the group's memory which lives in the historical repositories for a project. Hassan and Holt propose mining source control repositories and attaching Historical Sticky Notes to each code dependency in a software system [26]. These notes record various properties concerning a dependency such as the time it was introduced, the name of the developer who introduced it, and the rationale for adding it. Using the historical sticky notes on the NetBSD system, a large open source operating system, many unexpected dependencies could be easily explained and rationalized.

3.2 Propagating Changes

Change propagation is the process of propagating code changes to other entities of a software system to ensure the consistency of assumptions in the system after changing an entity. For example, a change to an interface may require the change to propagate to all the components which use that interface. The propagation of this change would ensure that both the interface and entities using it have a consistent set of assumptions. Change propagation plays a central role in software development. However current practices for propagating software changes rely heavily on human communication, and the knowledge of experienced developers. Many hard to find bugs are introduced by developers who did not notice dependencies between entities, and failed to propagate changes correctly.

Instead of using traditional dependency graphs to propagate changes, we could make use of the historical co-changes. The intuition is that entities co-changing frequently in the past are very likely to co-change in the future. This is similar in spirit to how retailers suggest other products for customers – Customers, who bought milk, bought cereal 99% of the time. Zimmermann *et al.* [68], Ying *et al.* [66], Shirabad [59], Hassan and Holt [25] show that suggestions based on historical co-changes are on average accurate 30% of the time (i.e., precision), and can correctly propose 44% of the entities which must co-change (i.e., recall). Recent results by Malik and Hassan show a precision of 64% and a recall of 78% could be achieved using more advanced data mining techniques [45]. Hassan and Holt also demonstrate that historical dependencies outperform traditional dependency information when propagating changes for several open source projects [25]. Kagdi *et al.* demonstrate the applicability of historical inforamtion in propagating changes from code to documentation entities where there are no structural code dependencies [33].

3.3 Predicting and Identifying Bugs

Predicting the occurrence of bugs remains one of the most active areas in software engineering research. Using this knowledge, managers can allocate testing resource appropriately, developers can review risky code more closely, and testers can prioritize their testing efforts. A substantial number of complexity metrics have been proposed over the years. The intuition being that complex code is likely to be buggy. However, results by Graves *et al.* [22] on a commercial system and by Herraiz *et al.* on a large sample of open source projects [28] show that most complexity metrics correlate well with LOC. Instead Graves *et al.* indicate that the two of the best predictors of bugs are prior bugs and prior changes, i.e., code that has bugs in the past is likely to have bugs in the future and bugs are not likely to appear in unchanged code. Moreover, Graves *et al.* [22] show that recent bugs and changes have a higher effect on the bug potential of a code over older changes. The importance of process metrics from historical repositories in predicting bugs over traditional complexity metrics has as well been recently shown by Moser *et al.* for the Eclipse open source project [52].

Instead of predicting the number of future bugs, practitioners desire tools that would flag the location of bugs in their current code base so they can fix these bugs. Unfortunately the specifications of large software applications rarely exist, so comparing an application against its expected and specified behavior using traditional static analysis techniques is usually not feasible in practice. Instead researchers have adopted novel techniques which analyze large amount of data about a software application to uncover the dominant behavior or patterns and to flag variations from that behavior as possible bugs. By analyzing the source code of an application, tools such as Coverity [16], can uncover undocumented correctness rules such as "acquire lock L before modifying x" or "do not block when interrupts are disabled". The tools then flag any deviations of these rules in the source code. These deviations are likely bugs. Several types of rules could be mined from the source code: Coverity and PR-Miner [41] mine function-pairing rules; Daikon [17] and DIDUCE [23] mine variable-value invariants by analyzing execution traces; MUVI mines variable-pairing rules [44]; and AutoISES mines function and variable access rules to detect security violations [61]. Recent work by Jiang *et al.* applies many of the aforementioned techniques to uncover load testing problems for large enterprise applications [31]. A load test involves the repetitive execution of similar requests by an application; the proposed approach mines the execution logs to uncover dominant rules and deviation from these rules.

Other tools such as CP-Miner [40] can flag bugs by recognizing deviations in mined patterns for renaming variables when clonning (i.e., copy-and-paste) code.

Using historical code changes, DynaMine [43] uncovers function-pairing rules. For example, if addListener() and removeListener() are always added together to the code, then a change were addListener() is added without removeListener() is likely a buggy change. Williams and Hollingsworth use historical changes to refine the order of warnings produced by static code checkers [63]. For example, warnings for missing to check the return-value for a called function are ordered based on examining prior code changes – If a developer in the past added a check for the return-value for a particular function, then the importance of checking the return of that function is asserted, and warnings related to missing check are moved higher in the warning list.

3.4 Understanding Team Dynamics

Many large projects communicate through mailing lists, IRC channels, or instant messaging. These discussions cover many important topics such as future plans, design decisions, project policies, and code or patch reviews. These discussions represent a rich source of historical information about the inner workings of large projects. These discussions could be mined to better understand the dynamics of large software development teams.

In many open source projects, project outsiders can submit code patches to the mailing list. However outsiders cannot commit code directly to the source control repository until they are invited to be part of the core group of developers of a project. Inviting an outsider too early may lead to inviting individuals that are not well-qualified or may not fit well with the group. Inviting an outsider too late may lead to the outsider losing interest in the project and the project losing a valuable core developer. Bird *et al.* study the usual timelines for inviting developers to the core group in open source projects by mining information from source code repositories and mailing lists [9].

In addition to uncovering the process for inviting developers, mailing lists discussions could uncover the overall morale of a development team with developers using more optimistic words when they feel positive about the progress of the project and using negative words when they are concerned about the state of the project. Rigby and Hassan used a psychometric text analysis tool to analyze the mailing lists discussions to capture the overall morale of a development team before and after release time for the Apache web server project [56].

Users and developers are continuously logging bugs in the bug repository of large software projects. Each report must be triaged to determine if the report should be addressed, and to which developer it should be assigned. Bug triage is a time-consuming effort requiring extensive knowledge about a project and the expertise of its developers. Anvik and Murphy speed up the bug triage efforts by using

prior bug reports to determine the most suitable developers to which a bug should be assigned [5].

3.5 Improving the User Experience

Michail and Xie propose a Stabilizer tool which mines reported bugs and execution logs to prevent an application from crashing [47]. When a user attempts to perform an action which has been reported by others to be buggy, the Stabilizer tool presents a warning to the user who is given the opportunity to abort the action. This approach permits users to use buggy applications while they wait for developers to fix bugs.

Mockus *et al.* study the quality of a software application as perceived by its users [51]. Instead of studying the quality of the source code, they mine data captured by project monitoring and tracking infrastructures as well as customer support records to determine the expected quality of a software application. They find that the deployment schedule, hardware configurations, and software platform have a significant effect on the perceived quality of an application, increasing the probability of observing a software failure by more than 20 times.

3.6 Reusing Code

Code reuse is an important activity in modern software development. Unfortunately, large code libraries are usually not well-documented, and have flexible and complex APIs which are hard to use by non-experts. Several researchers have proposed tools and techniques to mine code repositories to help developers reuse code. The techniques locate uses of code such as library APIs, and attempt to match these uses to the needs of a developer working on a new piece of code. For example, Mandelin *et al.* develop a technique which helps a developer write the code needed to get access to a particular object [46], while Xie and Pei [65] propose a technique to help a developer write the setup and tear down code needed to use a library method.

3.7 Automating Empirical Studies

A major contribution of the MSR field, is the automation of many of the activities associated with empirical studies in software engineering. The automation permits the repetition of studies on a large number of subject and the ability to verify the generality of many findings in these studies. For example, recent work by Robles *et al.* [57] showed that the growth of 18 large open software follows a usually linear or superlinear trend. This work contradicts the fourth law of software evolution which was proposed by Lehman and colleagues [39] based on a small sample of industrial system. The MSR automation enables the verification of the generality of prior findings.

Common wisdom and literature about code cloning indicate that cloning is harmful and has a negative impact on the maintainability of software systems. Kapser and Godfrey examine several large open source systems and show that cloning is often used as a principled engineering tool [34]. For example, cloning is often used to experiment with new features while ensuring the stability of the old code. New features are added in cloned code then re-integrated into the original code once the features are stabilized.

4 Opportunities in the Road Ahead

In the last section, we gave a brief survey of the impact of MSR on many important dimensions in software research and practice. The MSR field is very fortunate in that the cost of experimenting with MSR techniques is usually low – the data needed to perform MSR research and to demonstrate the value of adopting many of the MSR findings is readily available as it is collected by projects for other purposes. In the road ahead there exists many opportunities for the MSR field to demonstrate the strategic importance of software repositories and the benefit of transforming static software repositories into active ones which could support and automate many daily decision-processes in modern software development organizations. We present below several areas opportunities and discuss the challenges associated with these opportunities and work done within the MSR community to tackle some of these challenges.

4.1 Taming the Complexity of MSR

MSR techniques remain as advanced techniques with a large barrier of entry due to the complexities associated with data extraction and analysis. Although the data needed for MSR is readily available, the data is not collected with mining in mind. For example, commonly used source control repositories, such as CVS and subversion, track changes at the file level instead of tracking changes at the code entity level (e.g., functions and classes). And CVS, the most used source control repository, does not track the fact that several files have co-changed together or the purpose of a change, e.g., to fix a bug or to merge a code branch.

In the next few years, MSR researchers should focus on lowering the barrier of entry into the MSR field. Lowering the barrier of entry will bring in researchers from several other domains and wider experience, and will raise the diversity of the important problems that are being investigated using software repositories. MSR researchers should work on documenting best practices for mining repository data, and on providing access to mining tools and already mined data in well-defined exchange formats. Early work in exchange formats, such as [36], is still not well-adopted Nevertheless there are currently a few toolsets (e.g., [42]) and datasets (e.g., [2, 19]) that are available for others to use. The following are a few of the challenges that researchers must address by providing toolkits and advice for others to overcome these challenges.

Simplifying the Extraction of High-Quality Data. Many heuristics are used to extract data from code repositories (e.g., [24, 67, 18, 49]), from email repositories (e.g., [8, 56]), and from run-time repositories (e.g., [31]). Heuristics are used to deal with un-compilable code in a code repository. Heuristics are used to map a user's email to a single individual since users do not have unique emails over the years or even within the same day (sending emails from home and work or school email addresses). Researchers should closely examine, document, and study the correctness of the used heuristics. Toolkits to help others extract data with limited knowledge about these heuristics are needed. However building tools and extracting data from repositories are time consuming and challenging tasks. There is currently a lack of proper ways to acknowledge the contributions of researchers who build tools and provide or share extracted data. The contributions of these researchers should be acknowledged as an important and essential contribution to the MSR field for these researchers and others to tackle this challenge.

Dealing with Skew in Repository Data. Often the data available in software repositories exhibits a large amount of noise and skew in it. For example, the count of changes and bugs to files tends to have high skew in it with a small portion of files having most of the bugs or changes in them. This large skew requires special attention when using traditional data mining algorithms. For example, decision tree learners have a high tendency to simply return the most common category in the data as a prediction when there most-common-category occurs at a very high rate. More robust algorithms and data re-sampling techniques should be adopted [45].

Another example of skew exits in source control repositories with a small number of changes involving most of the files in a project. For example, each year all files of a project may be changed together in one large change to update the copyright year at the top of each file. The use of such data may lead to incorrect results and conclusions. MSR researchers should closely study the noise and skew in the data and better understand their effect on the analysis. Guidelines, techniques and tools are needed to detect noise and skew, and to accommodate them in the analysis. Visualization techniques, such as [15, 38, 64] are essential in helping spot noise and are important tools when mining software repositories. Detailed statistical sampling or manual (e.g., [29]) analysis may be needed to better understand the characteristics of the noise and whether it should be included in the analysis.

Scaling MSR Techniques to Very Large Repositories. Most MSR techniques have been demonstrated on large scale software systems. However, the size of data available for mining continues to grow at a very rapid rate. More intelligent techniques are needed to handle such large amount of data. These techniques must tackle the size and the age of the data. For example, when analyzing historical repositories, techniques should explore assigning more weight to recent data over older data [22].

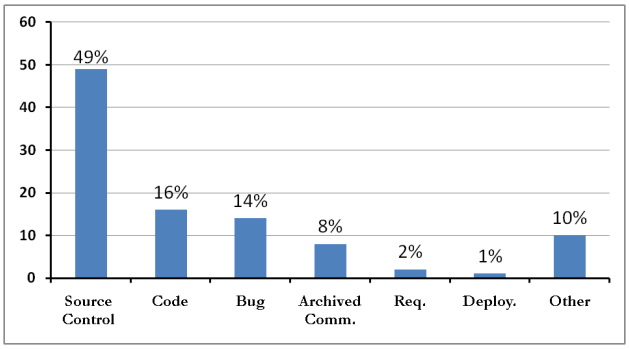

Figure 1. Repositories in MSR publications

Improving the Quality of Repository Data. Researchers should provide guidelines and tools to practitioners who are interested in supporting future mining efforts of their repositories by improving the data entered in their repository. Lexical heuristics used to determine the rationale for a change [49] would not be needed, if the User Interface of source control repository tool would permit a developer to select the purpose of a change from a drop-down menu. Similarly, heuristics to determine the actual change which fixed a particular bug [37] would no longer be required if developers would simply enter the bug-id when committing their bug-fix change.

4.2 Going Beyond Code and Bugs

An analysis of the publications at the MSR venue over the last five years along various types of repositories (*see* Figure 1) shows that a large amount (∼80%) of the published work focuses on source code and bug related repositories. We believe that this is primarily due to the fact that the used repositories (e.g., bug repositories or source control repositories) are commonly available and the structured nature of source code and bug reports which eases the analysis. The Figure also shows that documentation repositories (e.g., requirements) are rarely studied primarily due to the limited availability of such repositories. The Figure is derived by examining the full and short papers published in the proceedings of MSR from 2004 to 2008. The Figure counts all types of repositories used by a particular paper. For example, a paper may use data from a source control repository and link it with archived project communication

(e.g., emails), so we mark the paper as using both types of repositories.

MSR research should expand its analysis beyond structured data and beyond data in single repository, while tackling the following challenges.

Exploring Non-Structured Data. Although program data is usually structured, repositories contain unstructured data such as archived communication repositories which contain natural language text [56]. Similarly deployment logs in practice usually do not follow a strict structure [31]. Although Figure 1 shows that there has been a large focus on using source control and bug repositories in papers, their use has shifted in recent years to study other non-code aspects of these repositories such as analyzing the social network of software developers who committed changes or fixed bugs to these repositories. The study of non-structured data in source control repositories and non-structured repositories (e.g., archived communications) continues to rise in popularity with many papers examining the social and technical aspects of software projects. As software engineering researchers, our knowledge and experience with techniques to analyze non-structured data is limited so we should collaborate closely with experienced researchers, such as as social scientists.

Linking Data Between Repositories. A large amount of research uses data in a single repository. However the use and linking of data across different repositories can help in improving the quality of the data [37] and in providing practitioners with a more complete view of the project [12]. Techniques which could accurately link a bug report, to the email discussions about it, to the logs for deployments exhibiting it, to test cases which verify it, and to the actual change which introduced the bug and the change which fixed it would be very valuable in improving the type of analysis done in the field. Research should explore the benefits and challenges of linking data between repositories.

Seeking Non-Traditional Repositories. Figure 1 shows that around 10% of all publications make use of non-traditional repositories. Examples of such repositories are: repositories of build warnings or test results, repositories of programs in large software distributions (e.g., Linux distributions), and IDE interaction history. Some of these repositories, such as IDE interaction history, may not be currently available; while other repositories,e.g., build warnings or test results, may not be commonly available. The work presented at MSR provides a critical analysis of the risks and benefits of building or using such repositories in supporting software research and practice. As research progresses in the MSR field, researchers and practitioners should explore other non-traditional (i.e., not-commonly available) repositories and demonstrate the importance of these repositories so others can better understand the value of creating and maintaining such repositories for their projects.

Understanding the Limitations of Repository Data. Repository data cannot be used to conclude causation instead it can only show correlation. MSR findings must be investigated more closely within the context of the studied project or system. Project and system context are very important to reveal the true cause for particular findings. For example, although an analysis of historical repositories may show that particular developers are more likely to perform buggy changes. This may be due to the fact that they are usually assigned more complex changes and not due to the skill level of these developers.

Moreover, findings may not generalize across projects [53] and the use of repositories varies between different projects. So researchers should closely examine the project culture to better understand the use of the repositories before reaching conclusions. For example, in open source projects a limited number of developers are given commit rights and they commit code for other contributors. Therefore an analysis of the source control repository may incorrectly indicate that there are a small number of contributors to open source projects whereas in reality the number of contributors is much larger [21, 48]. In short, the limitation of repository data should be closely examined and communicated when presenting the results of mining research to avoid the misinterpretation of findings.

4.3 Showing the Value of Repositories

MSR researchers should continue to show and demonstrate the value of data in software repositories and the benefits of MSR techniques in helping practitioners in their daily activities. With practitioners seeing the value of the MSR field, they are more likely to consider adopting MSR techniques in practice. They are also more willing to work on improving the quality of the data stored in software repositories to ease future data extraction efforts and to improve the quality of results of mining efforts. The following are a few of the challenges that MSR researchers must tackle to demonstrate the value of software repositories.

Understanding the Needs of Practitioners. The effectiveness of MSR techniques should be explored relative to the needs of practitioners. We should aim to address problems that are relevant and important to them. For example, although there exists many techniques to predict buggy files in large software projects, the value of such predictions may be low for developers who are usually well-aware of the most buggy parts of their system.

Similarly bug prediction techniques which predict the incidence of bugs at the file level, may be too coarse of a level for practitioners to adopt as they may not add much to their current knowledge. Metrics which could be mapped to actual time and money savings, if possible, are highly desired.

Studying The Performance of Techniques in Practice. The effectiveness of MSR techniques is often demonstrated using historical repository data. Once these techniques are adopted, it is not clear how these proposed techniques would affect the daily activities of practitioners which in turn would affect the data stored in the repository. For example, research has shown that techniques, which use historical code co-changes to predict other entities which must be co-changed, perform well. However if such techniques are adopted in practice, developers may start relying too much on these techniques to guide them. This high dependence on the technique will affect the historical co-change data which the techniques themselves depend on to perform well.

Showing the Practical Benefit of MSR Techniques. Researchers must not only demonstrate the statistical benefit and improvement of MSR techniques over traditional techniques. They must also discuss the practical benefit and cost of their techniques. For example, techniques that require a great deal of manual work and maintenance may be hard to adopt in practice, even if they outperform other techniques unless the manual work is a one-time effort.

The performance of most MSR techniques is measured by demonstrating the performance of the techniques at a particular point of time. The maintenance efforts needed to ensure that these techniques keep on performing well is not studied. A technique which mines historical repositories may perform well when first adopted. But as the amount of historical data increases, the technique performance may suffer and the technique may require some calibration. Techniques which can auto-calibrate or which can at least warn users that they need re-calibration are needed. Techniques which require minimal intervention once adopted are highly desirable. Such area of work has not yet to be explored in the MSR field. However, the importance of such work is well-recognized in traditional data mining. For example when mining consumer data, the shift of demographics or consumers perception should be accounted for.

Evaluating Techniques on Non Open Source Systems. MSR researchers continue to demonstrate their techniques on open source system due to their accessability and availability. The generalization of findings and techniques to commercial, non-open source systems, has not be studied. Unfortunately, access to commercial repositories still remains limited.

4.4 Easing the Adoption of MSR

Although the value of MSR techniques may have been demonstrated, there are several challenges preventing the adoption of MSR techniques by practitioners and researchers. Müller and colleagues have over the past few years explored many of the challenges associated with adopting software engineering research in practice (e.g., [6]). We discuss below a few of these challenges from an MSR perspective and we comment on work done in the MSR community to address some of these challenges.

Simplifying Access to Techniques. MSR Researchers should simplify the resources and tools needed for practitioners to experiment and use MSR techniques on their repositories. One option is to integrate MSR techniques into current toolsets instead of establishing MSR-specific toolsets. Fortunately, modern development environments such as Eclipse offer APIs to extend them. HATARI [60], Hipikat [12], Myln [35], and eROSE [68] are examples of MSR research along this direction. These tools integrate within development environments and require minimal effort for practitioners to use. Another option is to consider web service for MSR techniques where repositories can be uploaded for analysis. One possibility is to use a few large open source projects as guinea pigs, to demonstrate the functionality of such services and the benefits of MSR techniques to practitioners.

Helping Practitioners Make Decisions MSR techniques should not aim for full automation instead they should aim to create a synergy between practitioners and MSR techniques. Surprisingly full automation is not always the most desired option for practitioners. Practitioners prefer techniques that support their decision-making process instead of replacing them [11]. Practitioners also prefer simple and easy to understand and rationalize models (e.g., decision tree) over better-performing yet complex models (e.g., genetic algorithms). The simplicity of the models help practitioners in rationalizing the output of MSR techniques, and in gaining buy-in by other shareholders in large projects.

5 Conclusion

Software repositories have traditionally been used for archival purposes. The MSR field has shown that these repositories could be mined to uncover useful patterns and actionable information about software systems and projects. MSR researchers have proposed techniques which augment traditional software engineering data, techniques and tools,

in order to solve important and challenging problems, such as identifying bugs, and reusing code, which practitioners must face and solve on a daily basis.

In this paper, we presented a brief history of the MSR field and discussed several recent achievements and results of using MSR techniques to support software research and practice. We then explored the various opportunities and challenges that lie in the road ahead for this important and emerging field while highlighting work done in the MSR community to address some of these challenges. For up-to-date information about the MSR field, please refer to `http://msrconf.org/`.

Acknowledgements

The author thanks Stephan Diehl, Daniel German, Zhen Ming Jiang, Tom Zimmermann, and Ying Zou for their suggestions on improving the paper and for sharing their thoughts and ideas. The author gratefully acknowledges the contributions of the members of the MSR community for their work and input in helping establish and grow the MSR community as an important part of Software Engineering.

The MSR community as a whole acknowledges the significant contributions from the open source community who assisted our community in understanding and acquiring their valuable software repositories. These repositories were essential in progressing the state of research in the MSR field and Software Engineering.

References

[1] Summary of Sarbanes-Oxley Act of 2002. `http://www.soxlaw.com/`.

[2] The PROMISE repository. `http://promisedata.org/`.

[3] *Proceedings of the 25th International Conference on Software Engineering, May 3-10, 2003, Portland, Oregon, USA*. IEEE Computer Society, 2003.

[4] *Fourth International Workshop on Mining Software Repositories, MSR 2007 (ICSE Workshop), Minneapolis, MN, USA, May 19-20, 2007, Proceedings*. IEEE Computer Society, 2007.

[5] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In Osterweil et al. [54], pages 361–370.

[6] R. Balzer, J. H. Jahnke, M. Litoiu, H. A. Müller, D. B. Smith, M.-A. D. Storey, S. R. Tilley, and K. Wong. 3rd International Workshop on Adoption-centric Software Engineering (ACSE). In *ICSE* [3], pages 789–790.

[7] V. R. Basili and B. Perricone. Software Errors and Complexity: An Empirical Investigation. *Communications of the ACM*, 27(1):42 – 52, 1984.

[8] C. Bird, A. Gourley, P. Devanbu, M. Gertz, and A. Swaminathan. Mining Email Social Networks. In *Proceedings of the 3rd International Workshop on Mining Software Repositories*, Shanghai, China, May 2006.

[9] C. Bird, A. Gourley, P. T. Devanbu, A. Swaminathan, and G. Hsu. Open Borders? Immigration in Open Source Projects. In *MSR* [4], page 6.

[10] A. Chen, E. Chou, J. Wong, A. Y. Yao, Q. Zhang, S. Zhang, and A. Michail. CVSSearch: Searching through Source Code Using CVS Comments. In *Proceedings of the 17th International Conference on Software Maintenance*, pages 364–374, Florence, Italy, 2001.

[11] J. R. Cordy. Comprehending Reality - Practical Barriers to Industrial Adoption of Software Maintenance Automation. In *IWPC*, pages 196–206. IEEE Computer Society, 2003.

[12] D. Cubranic, G. C. Murphy, J. Singer, and K. S. Booth. Hipikat: A Project Memory for Software Development. *IEEE Trans. Software Eng.*, 31(6):446–465, 2005.

[13] S. Diehl, H. Gall, and A. E. Hassan, editors. *Proceedings of the 2006 International Workshop on Mining Software Repositories, MSR 2006, Shanghai, China, May 22-23, 2006*. ACM, 2006.

[14] S. G. Eick, C. R. Loader, M. D. Long, S. A. V. Wiel, and L. G. Votta. Estimating Software Fault Content Before Coding. In *Proceedings of the 14th International Conference on Software Engineering*, pages 59–65, Melbourne, Australia, May 1992.

[15] S. G. Eick, J. L. Steffen, and E. E. Sumner. Seesoft-A Tool For Visualizing Line Oriented Software Statistics. *IEEE Trans. Software Eng.*, 18(11):957–968, 1992.

[16] D. R. Engler, D. Y. Chen, and A. Chou. Bugs as Inconsistent Behavior: A General Approach to Inferring Errors in Systems Code. In *SOSP*, pages 57–72, 2001.

[17] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon System for Dynamic Detection of Likely Invariants. *Sci. Comput. Program.*, 69(1-3):35–45, 2007.

[18] M. Fischer, M. Pinzger, and H. Gall. Populating a Release History Database from Version Control and Bug Tracking Systems. In *ICSM*, pages 23–. IEEE Computer Society, 2003.

[19] FLOSSmole. Available online at `http://ossmole.sourceforge.net/`.

[20] H. Gall, K. Hajek, and M. Jazayeri. Detection of Logical Coupling Based on Product Release History. In *Proceedings of the 14th International Conference on Software Maintenance*, Bethesda, Washington D.C., Nov. 1998.

[21] D. M. Germán. An Empirical Study Of Fine-Grained Software Modifications. *Empirical Software Engineering*, 11(3):369–393, 2006.

[22] T. L. Graves, A. F. Karr, J. S. Marron, and H. P. Siy. Predicting Fault Incidence Using Software Change History. *IEEE Transactions on Software Engineering*, 26(7):653–661, 2000.

[23] S. Hangal and M. S. Lam. Tracking Down Software Bugs Using Automatic Anomaly Detection. In *ICSE*, pages 291–301. ACM, 2002.

[24] A. E. Hassan. *Mining Software Repositories to Assist Developers and Support Managers*. PhD thesis, University of Waterloo, 2004.

[25] A. E. Hassan and R. C. Holt. Predicting Change Propagation in Software Systems. In *Proceedings of the 20th International Conference on Software Maintenance*, Chicago, USA, Sept. 2004.

[26] A. E. Hassan and R. C. Holt. Using Development History Sticky Notes to Understand Software Architecture. In *Proceedings of the 12th International Workshop on Program Comprehension*, Bari, Italy, June 2004.

[27] A. E. Hassan, A. Mockus, R. C. Holt, and P. M. Johnson. Guest Editor's Introduction: Special Issue on Mining Software Repositories. *IEEE Trans. Software Eng.*, 31(6):426–428, 2005.

[28] I. Herraiz, J. M. González-Barahona, and G. Robles. Towards a Theoretical Model for Software Growth. In *MSR* [4], page 21.

[29] A. Hindle, D. M. Germán, and R. C. Holt. What Do Large Commits Tell Us?: A Taxonomical Study Of Large Commits. In A. E. Hassan, M. Lanza, and M. W. Godfrey, editors, *MSR*, pages 99–108. ACM, 2008.

[30] J. P. Hudepohl, S. J. Aud, T. M. Khoshgoftaar, E. B. Allen, and J. Mayrand. Emerald: Software Metrics and Models on the Desktop. *Computer*, 13(5), 1996.

[31] Z. M. Jiang, A. E. Hassan, P. Flora, and G. Hamann. Automatic Identification of Load Testing Problems. In *Proceedings of the 24th International Conference on Software Maintenance*, Beijing, China, Sept. 2008.

[32] H. H. Kagdi, M. L. Collard, and J. I. Maletic. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *Journal of Software Maintenance*, 19(2):77–131, 2007.

[33] H. H. Kagdi, J. I. Maletic, and B. Sharif. Mining software repositories for traceability links. In *ICPC*, pages 145–154. IEEE Computer Society, 2007.

[34] C. Kapser and M. W. Godfrey. "Cloning Considered Harmful" Considered Harmful. In *WCRE*, pages 19–28. IEEE Computer Society, 2006.

[35] M. Kersten and G. C. Murphy. Mylar: A Degree-of-interest Model for IDEs. In M. Mezini and P. L. Tarr, editors, *AOSD*, pages 159–168. ACM, 2005.

[36] S. Kim, T. Zimmermann, M. Kim, A. E. Hassan, A. Mockus, T. Gîrba, M. Pinzger, J. Whitehead, and A. Zeller. TA-RE: An Exchange Language for Mining Software Repositories. In Diehl et al. [13], pages 22–25.

[37] S. Kim, T. Zimmermann, K. Pan, and J. Whitehead. Automatic Identification of Bug-Introducing Changes. In *ASE*, pages 81–90. IEEE Computer Society, 2006.

[38] M. Lanza and S. Ducasse. Polymetric Views – A Lightweight Visual Approach to Reverse Engineering. *IEEE Trans. Software Eng.*, 29(9):782–795, 2003.

[39] M. M. Lehman, J. F. Ramil, P. Wernick, D. E. Perry, and W. M. Turski. Metrics and Laws of Software Evolution - The Nineties View. In *IEEE METRICS*, pages 20–. IEEE Computer Society, 1997.

[40] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code. *IEEE Trans. Software Eng.*, 32(3):176–192, 2006.

[41] Z. Li and Y. Zhou. PR-Miner: Automatically Extracting Implicit Programming Rules and Detecting Violations in Large Software Code. In Wermelinger and Gall [62], pages 306–315.

[42] LibreSoft tools web site. Available online at `http://tools.libresoft.es/`.

[43] V. B. Livshits and T. Zimmermann. DynaMine: Finding Common Error Patterns By Mining Software Revision Histories. In Wermelinger and Gall [62], pages 296–305.

[44] S. Lu, S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. A. Popa, and Y. Zhou. MUVI: Automatically Inferring Multi-Variable Access Correlations and Detecting Related Semantic and Concurrency Bugs. In T. C. Bressoud and M. F. Kaashoek, editors, *SOSP*, pages 103–116. ACM, 2007.

[45] H. Malik and A. E. Hassan. Supporting Software Evolution Using Adaptive Change Propagation Heuristics. In *Proceedings of the 24th International Conference on Software Maintenance*, Beijing, China, Sept. 2008.

[46] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid Mining: Helping to Navigate the API Jungle. In V. Sarkar and M. W. Hall, editors, *PLDI*, pages 48–61. ACM, 2005.

[47] A. Michail and T. Xie. Helping Users Avoid Bugs in GUI Applications. In Roman et al. [58], pages 107–116.

[48] A. Mockus, R. T. Fielding, and J. D. Herbsleb. A Case Study of Open Source Software Development: the Apache Server. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 263–272, Limerick, Ireland, June 2000. ACM Press.

[49] A. Mockus and L. G. Votta. Identifying Reasons for Software Changes using Historic Databases. In *ICSM*, pages 120–130, 2000.

[50] A. Mockus, D. M. Weiss, and P. Zhang. Understanding and Predicting Effort in Software Projects. In *ICSE* [3], pages 274–284.

[51] A. Mockus, P. Zhang, and P. L. Li. Predictors Of Customer Perceived Software Quality. In Roman et al. [58], pages 225–233.

[52] R. Moser, W. Pedrycz, and G. Succi. A Comparative Analysis Of The Efficiency Of Change Metrics And Static Code Attributes For Defect Prediction. In Robby, editor, *ICSE*, pages 181–190. ACM, 2008.

[53] N. Nagappan, T. Ball, and A. Zeller. Mining Metrics to Predict Component Failures. In Osterweil et al. [54], pages 452–461.

[54] L. J. Osterweil, H. D. Rombach, and M. L. Soffa, editors. *28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006*. ACM, 2006.

[55] D. E. Perry and W. M. Evangelist. An Empirical Study of Software Interface Errors. In *Proceedings of the International Symposium on New Directions in Computing*, pages 32–38, Trondheim, Norway, Aug. 1985.

[56] P. C. Rigby and A. E. Hassan. What Can OSS Mailing Lists Tell Us? A Preliminary Psychometric Text Analysis of the Apache Developer Mailing List. In *MSR* [4], page 23.

[57] G. Robles, J. J. Amor, J. M. González-Barahona, and I. Herraiz. Evolution and Growth in Large Libre Software Projects. In *IWPSE*, pages 165–174. IEEE Computer Society, 2005.

[58] G.-C. Roman, W. G. Griswold, and B. Nuseibeh, editors. *27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA*. ACM, 2005.

[59] J. S. Shirabad. *Supporting Software Maintenance by Mining Software Update Records*. PhD thesis, University of Ottawa, 2003.

[60] J. Sliwerski, T. Zimmermann, and A. Zeller. HATARI: Raising Risk Awareness. In Wermelinger and Gall [62], pages 107–110.

[61] L. Tan, X. Zhang, X. Ma, W. Xiong, and Y. Zhou. AutoISES: Automatically Inferring Security Specifications and Detecting Violations. In *Proceedings of the 17th USENIX Security Symposium (USENIX Security '08)*, July-August 2008.

[62] M. Wermelinger and H. Gall, editors. *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005*. ACM, 2005.

[63] C. C. Williams and J. K. Hollingsworth. Automatic Mining of Source Code Repositories to Improve Bug Finding Techniques. *IEEE Trans. Software Eng.*, 31(6):466–480, 2005.

[64] J. Wu, R. C. Holt, and A. E. Hassan. Exploring Software Evolution Using Spectrographs. In *WCRE*, pages 80–89. IEEE Computer Society, 2004.

[65] T. Xie and J. Pei. MAPO: Mining API Usages From Open Source Repositories. In Diehl et al. [13], pages 54–57.

[66] A. T. T. Ying, G. C. Murphy, R. T. Ng, and M. Chu-Carroll. Predicting source code changes by mining change history. *IEEE Trans. Software Eng.*, 30(9):574–586, 2004.

[67] T. Zimmermann and P. Weißgerber. Preprocessing CVS Data for Fine-Grained Analysis. In *Proceedings of the 1st International Workshop on Mining Software Repositories*, Edinburgh, UK, May 2004.

[68] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller. Mining Version Histories to Guide Software Changes. *IEEE Trans. Software Eng.*, 31(6):429–445, 2005.