Mining Software Repositories to Assist Developers and Support Managers

Ahmed E. Hassan
Dept. of Electrical and Computer Engineering
University of Victoria
Victoria, Canada
ahmed@ece.uvic.ca

ABSTRACT

Software repositories (such as source control repositories) contain a wealth of valuable information regarding the evolutionary history of a software project.

This dissertation presents approaches and tools which mine and transform static record keeping software repositories to active repositories used by researchers to gain empirically based understanding of software development, and by practitioners to predict, plan and understand various aspects of their project. Our work is validated empirically using data based on over 60 years of development history for several open source projects.

1. INTRODUCTION

Historical information stored in software repositories provide a great opportunity to study large projects and products while not interfering with development processes and deadlines: *Source control systems* store changes to the source code as development progresses, *defect tracking systems* follow the resolution of software defects, and *archived communications* between project personnel record rationale for decisions throughout the life of a project. Such historical data is available for most software projects and represents a detailed and rich record of the historical development of a software system. Moreover, current software engineering research approaches and techniques can benefit from using such historical information. For example, historical information can assist developers in understanding the rationale for the current structure of a software system [6]. This dissertation explores mining data stored in software repositories in order to support software developers and managers in their endeavors to build and maintain complex software systems.

1.1 Early Research

Software repositories have primarily been used for historical record keeping activities such as retrieving old versions of the source code or tracking the status of a defect. A few studies have emerged that use this data to study various aspects of software development such as software architecture, code reuse, development process and developer motivation. Research by Gall *et al.* [2] has shown that software repositories can support developers changing legacy systems by pointing out hidden code dependencies. Chen *et al.* [1] have shown that historical information can assist developers in understanding large systems. Graves *et al.* [3] and Mockus *et al.* [8] demonstrated that historical change information can support management in predicting bugs and effort to ease the evolution of reliable software systems. These early studies highlighted the benefits of historical project data.

1.2 Personal Experience

Working as part of several industrial organizations, such as Research In Motion, IBM Research, and Nortel, the author found himself and other developers examining software repositories (such as source control systems), in an ad-hoc fashion, to clarify many of our concerns and understanding of a software system or to gauge the state of a software project, for example:

1. In the role of a head developer of a project, we were frequently asked for estimates on when a project is ready for release, about a project's expected reliability or concerning the need for testing resources – we adopted coarse estimates by examining the change history of the software system as stored in its source control repository.
2. In the role of a developer, we faced the daunting task of understanding large complex systems which were developed by others, enhanced by many and patched frequently to meet tight deadlines or critical emergencies – we found ourselves along with other developers falling back to the initial version of a complex piece of code to understand it. In many cases, the initial cut of a piece of code was easier to understand and was cleaner than the current code. Moreover, we often investigated prior changes to code segments to gain a better understanding of the rationale for their current complexity or to clarify design choices.

1.3 The Open Source Phenomena

The promising results obtained from early studies along with our personal industrial experience highlighted to us the potential of software repositories in supporting developers and managers working on large software systems. In order to pursue our research, we needed a large number of projects (*Guinea Pigs*) for which we could analyze their historical

22nd IEEE International Conference on Software Maintenance (ICSM'06)
0-7695-2354-4/06 $20.00 © 2006 IEEE

development records. The explosive growth of open source offered us the opportunity to study several large open source projects who keep their repositories accessible online for developers around the world to contribute to their project. Furthermore, most of the project's communication and development documentation is archived online. The large number of available projects and the ease of access to their history permitted us to empirically verify our proposed techniques and approaches, and to interpret our findings.

2. RESEARCH HYPOTHESIS

Early findings and our industrial experience lead to the formation of our research hypothesis. We believe that:

> *Software repositories contain a wealth of valuable information about the evolution of a software project. By mining such historical information, we can develop techniques and approaches to support developers and managers in their endeavors to build and maintain complex software systems.*

Throughout out the dissertation, we demonstrated the value of mining software repositories by studying and formalizing ad-hoc techniques adopted by practitioners who use historical records as part of their day to day job. In particular, we developed approaches and techniques that use the evolutionary history of software projects to assist:

- Developers:
  - in understanding legacy code and discovering the rationale behind the current software structure.
  - in ensuring that changes are propagated to the appropriate parts of the code.
- Managers:
  - in predicting faults in a software system.
  - in allocating their limited testing resources to the most appropriate parts of the software system.

We as well demonstrated that the mining process can be automated in order to robustly process the historical records for long lived software systems. By automating this process, we could study a large number of systems. By documenting and presenting our process, interested researchers and practitioners could easily apply or extend our proposed techniques, and could investigate other possible uses of the recovered data.

3. OVERVIEW OF THE DISSERTATION

Our dissertation work could be broken into three different parts. Each part focuses on the interests of particular roles (researchers, developers, and managers). For researchers, we presented techniques to automate the mining process of large historical repositories. For developers and managers, we presented techniques to help them understand, maintain, and manage large projects using historical information.

3.1 Extracting Information From Repositories

Although software repositories are available for most large software projects, the data stored in these repositories has rarely been the focus of software engineering research. We believe that this is mainly owing to the following hurdles:

1. The limited access to such repositories prevented researchers from studying them. Companies in many cases are not willing to offer researchers access to such detailed historical information about their software systems. Another possible source for repositories to study is academic projects. Unfortunately, software systems developed in academia tend to have a small number of developers, a short life span, and their development history is not as rich nor as interesting as the history of long lived industrial software systems.
2. The complexity of processing large repositories in an automated fashion hindered the adoption and integration of data from 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 recover data from software repositories.

With the advent of open source systems, researchers had access to rich repositories for large projects developed by hundreds of developers over extended periods of time. This lead to early research in mining software repositories which was based on open source projects [1]. The second hurdle concerning the complexity of extracting information from these large historical repositories remained. To overcome this hurdle, we proposed *evolutionary code extractors*. These extractors automatically recover the evolutionary history of software systems and represent it in a simple and easy to access format, hence the data becomes more accessible for researchers to investigate and integrate in their studies.

We designed and developed an evolutionary code extractor for the C programming language called C-REX. C-REX mines the data stored by a source control system as an example of a software repository. Whereas, most source control systems record changes to the code at the file level, C-REX traces changes to specific source code entities, such as functions, variables, or data type definitions. C-REX can then track details such as:

- Addition, removal, or modification of a source code entity such as adding or removing a function.
- Changes to dependencies between source code entities. For example, we can determine that a function no longer uses a specific variable or that a function now calls another function.

Furthermore, C-REX lexically analyzes the content of the change message attached to a modification to automatically classify modifications into three types: Fault Repairing modifications (FR), Feature Introduction modifications (FI), and General Maintenance modifications (GM). The recovered data is stored in a format that is easy to process by researchers with little knowledge about software repositories.

Using the data recovered by C-REX for several open source projects, we conducted a survey to investigate how practitioners make use of change messages and what type of information exists in them. We investigated the quality of automatic classifications done by C-REX. We also asked practitioners to compare change messages in open source systems to ones in commercial systems. In particular, we sought to answer questions such as:

- Do developers usually enter meaningful and descriptive change messages?

- Do developers monitor such messages and react to their content?
- Do developers make use of these message as they maintain and enhance code, or are they ignored?
- Can we automatically determine the type of a change as being a bug fix or a feature?

The findings of our survey suggest that change messages are a valuable resource which practitioners use to maintain and manage software projects. For example, practitioners use change messages to understand the code when they are fixing a bug. Moreover, change messages in open source projects are similar to messages that developers encounter in large commercial projects. An automated approach to determine the purpose of a change using the change message is likely to produce results similar to a manual analysis performed by professional developers. The results of our survey along with our ability to automate the mining process of software repositories encouraged us to investigate techniques and approaches to study and formalize practitioners' ad-hoc uses of repositories. The following two parts of our dissertation study the use of historical data derived from software repositories to support developers and managers.

3.2 Using Software Repositories to Support Developers

Developers maintaining large software projects need tools to assist them in changing the software system and understanding it. The cost of performing incorrect changes to a legacy system is very expensive since it will likely introduce bugs.

Dependency graphs have been proposed and used in many studies and maintenance activities to assist developers in understanding large software systems before they embark on modifying them to meet new requirements or to repair faults. Call graphs and data usage graphs are the most commonly used dependency graphs. These graphs show the current structure of the software system (*e.g.* In a compiler, an *Optimizer* function calling a *Parser* function). These graphs fail to reveal details about the structure of the system that are needed to gain a better understanding. For example, traditional call graphs cannot give the rationale behind an *Optimizer* function calling a *Parser* function.

In [6], we presented an approach which recovers valuable information from source control systems and attaches this information to the static dependency graph of a software system. We call this recovered information – *Source Sticky Notes*. We showed how to use these notes along with the software reflexion framework [9] to assist in understanding the architecture of large software systems. To demonstrate the viability of our approach, we applied it to understand the architecture of NetBSD – a large open source operating system.

In [5], we proposed using the historical project information to assess the claimed benefits of code maintenance tools and strategies. We presented the *Development Replay* (DR) approach which reenacts the changes stored in source control repositories using a proposed tool or strategy. We presented a case study where the DR approach is used to empirically assess and compare the effectiveness of several not-yet-existing tools which promise to assist developers in propagating code changes. The approach is illustrated through a case study for five large open source systems with over 40 years of development history.

3.3 Using Software Repositories to Support Managers

Managers of large projects need to prevent the introduction of faults, ensure their quick discovery, and their immediate repair while ensuring that the software can evolve gracefully to handle new requirements by customers. Moreover, managers endeavor with varying degrees of success to wisely allocate their limited testing and development resources to the most appropriate parts of the code. Bug prediction and resource allocation issues become non-trivial challenges which managers must face and resolve successfully. Unfortunately, in many cases managers' attempts to resolve these issues are based on ad-hoc techniques and rough approximations. Their success depends on their intuition, experience and chance.

To assist managers in predicting the occurrence of faults and improving the reliability of software systems, we used sound mathematical concepts from information theory, such as Shannon's Entropy, to present a novel view of complexity in software [4]. We proposed a complexity metric that is based on the process followed by developers to produce the code (*the code development process*) instead of on the code or the requirements. We conjectured that: *A chaotic code development process negatively affects its outcome, the software system, through the occurrence of faults.* We validated our conjecture empirically through case studies using data derived from the development process history of six large projects with over 60 years of development history. Our entropy measurements have statistically significant better accuracy in predicting the occurrence of faults than simply using the number of prior modifications to a subsystem or prior faults in it as predictors of faults. Using our complexity metric, managers are likely to avoid delays and faults in a project over time.

To assist managers in coping with the challenges of allocating their limited resources effectively, we presented an approach (*The Top Ten List*) which highlights the ten most susceptible subsystems to have a fault [7]. The list is updated dynamically as the development of a system progresses. Managers can focus testing resources to the subsystems suggested by the list. The Top list approach holds a lot of promise and value for practitioners, it provides a simple and accurate technique to assist them in managing their resources as they maintain large evolving software systems. In contrast to count based techniques which focus on predicting an absolute count of faults in a system over time, or classification based techniques which focus on predicting if a subsystem is fault prone or not, we focus on predicting the subsystem that are most likely to have a fault in them in the near future. For example, even though a subsystem may not be fault prone or may only have a few number of predicted faults, it may be the case that a fault will be discovered in it within the next few days or weeks. Or in another case, even though a fault counting based technique may predict that a subsystem has a large number of faults, the faults may be dormant faults that are not likely to cause concerns in the

near future. If we were to draw an analogy to our work and rain prediction, our prediction model focuses on predicting the areas that are most likely to rain in the next few days. The predicted rain areas may be areas that are known to be dry areas (*i.e.* not fault prone) or may be areas which aren't known to have large precipitation values (*i.e.* low predicted faults).

4. CONTRIBUTIONS

The *conceptual contributions* of this dissertation center around the development of techniques and approaches to demonstrate the value of mining software repositories in assisting managers and developers in performing a variety of software development, maintenance, and management activities. The *technical contributions* of this dissertation focus on the development of tools and the invention of techniques to robustly automate the mining process for large long lived software systems written in industrial languages such as C. The *empirical contributions* of this dissertation are the application of all proposed techniques and approaches on several long lived large open source projects. The particular contributions of this dissertation, notably those that have been published, include the following:

1. **Evolutionary Software Extractors:** We introduced the idea of an evolutionary extractor and advocated the need for such extractors to study and mine the evolutionary history of software projects from historical repositories such as source control repositories. We presented the implementation challenges and techniques for an evolutionary code extractor for the C language (C-REX).
2. **Source Sticky Notes:** We proposed the benefits of attaching historical information to each dependency in a software system. We showed that these notes are useful in speeding up and automating the software architecture understanding process.
3. **Development Replay Approach:** The DR approach reenacts the development history of a software project using the changes stored in source control repositories. This approach permits us to empirically assess the effectiveness of not-yet-adopted or not-yet-existing code maintenance tools and strategies.
4. **Software Development Chaos:** We conjectured that a chaotic or complex development process negatively affects its outcome, the software system. We proposed and validate the benefits of a complexity metric that is based on the process followed by software developers in order to produce the code instead of on the code or the requirements.
5. **Top Ten List:** We introduced the notion that not all bugs are created equal, instead managers are more concerned about bugs that are likely to occur in the near future versus faulty code that is not likely to have faults appear in it for some time. We proposed metrics and models to measure traditional bug prediction techniques using such notion and ideas.

5. CONCLUSIONS

Our work contributes to software engineering and maintenance by showing that software repositories contain a wealth of useful information that could be easily mined and integrated with several software development practices in order to assist developers and managers. We hope this work will encourage academic researchers to explore integrating historical information in their work, and will entice practitioners to consider the potential of their repositories which are currently mainly used for static record keeping purposes.

The field of Mining Software Repositories is maturing thanks to the rich, extensive, and easily accessible repositories for open source projects. We believe the field is likely to take a central and important role in supporting software development practices and software engineering research. During the course of this dissertation, we proposed and co-organized workshops on Mining Software Repositories (MSR) at the International Conference on Software Engineering (ICSE). The workshops (http://msr.uwaterloo.ca) were very successful and were the most attended workshops at ICSE for the last three years. We as well co-edited on the MSR topic a special Issue of the IEEE Transactions on Software Engineering (TSE). The issue had the largest submissions in the history of the TSE. It received over 15% of all the submissions to the TSE in 2004.

6. REFERENCES

[1] 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.

[2] 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.

[3] 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.

[4] A. E. Hassan and R. C. Holt. Studying The Chaos of Code Development. In *Proceedings of the 10th Working Conference on Reverse Engineering*, Victoria, British Columbia, Canada, Nov. 2003.

[5] 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.

[6] 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.

[7] A. E. Hassan and R. C. Holt. The Top Ten List: Dynamic Fault Prediction. In *Proceedings of the 21st International Conference on Software Maintenance*, Budapest, Hungary, Sept. 2005.

[8] A. Mockus, D. M. Weiss, and P. Zhang. Understanding and predicting effort in software projects. In *Proceedings of the 25th International Conference on Software Engineering*, pages 274–284, Portland, Oregon, May 2003.

[9] G. C. Murphy, D. Notkin, and K. Sullivan. Software Reflexion Models: Bridging the Gap Between Source and High-Level Models. In *Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 18–28, New York, NY, Oct. 1995. ACM.