Software-Change Prediction: Estimated+Actual

Huzefa Kagdi and Jonathan I. Maletic
Department of Computer Science
Kent State University
Kent Ohio 44242
{hkagdi, jmaletic}@cs.kent.edu

Abstract

The authors advocate that combining the estimated change sets computed from impact analysis techniques with the actual change sets that can be recovered from version histories will result in improved software-change prediction. An overview of both impact analysis (IA) and mining software repositories (MSR) is given. These are compared and a discussion of their expressiveness and effectiveness is presented. A framework is proposed to integrate these two approaches for software-change prediction.

1. Introduction

Software artifacts such as source code and design documents are produced in an inherently incremental manner via continuous change. This makes software changes an integral part of the software evolution process [20]. Arguably, the complexity of software evolution is the complexity of software changes. The process and methodology supporting software changes are a decisive factor between the sustained high-quality evolution and the premature retirement of a software system. Therefore, it is imperative to devise methodologies to effectively estimate, plan for, and realize software changes. Software-change prediction is one of the essential activities with regards to supporting software changes.

There are two approaches that are investigated in the research community for software-change prediction. The investigations in *Impact Analysis (IA)* are among early efforts that recognized software-change management as an important activity of software maintenance. Given a proposed change in a software artifact, impact analysis estimates the other software artifacts that are also likely to change by analyzing the current version of a system. However, impact analysis takes a uni-version view to software-change prediction [2]. That is, only a single version of a system is analyzed to estimated changes due to a proposed change. Impact analysis does not leverage past predictions. Moreover, it is non-adaptive as the estimations are seldom refined with regards to the actual changes that take place.

Mining Software Repositories (MSR) is a relatively newer (and growing) area of research that supports software-change prediction from a historical perspective. MSR takes a multi-version view to software-change prediction. In MSR, past versions of the software artifacts are analyzed to uncover pertinent information and trends of software changes that are then used to predict changes. It is solely based on historical records of actual past changes.

In this paper, we evaluate these two approaches and advocate for their combined use for the purpose of software-change prediction. Our, preliminary interest is to compare them with regards to their *expressiveness* and *effectiveness*. The expressiveness of a paradigm is discussed in terms of the granularity of predicted artifacts (e.g., file, functions, and variables). The effectiveness of a paradigm is discussed in terms of accuracy (i.e., precision and recall). This examination is a step towards answering our larger research question as to what are the exclusive and synergistic benefits of the two paradigms to improve software-change prediction. We present some specific questions on this issue and propose an infrastructure to support examination of these questions.

The rest of the paper is organized as follows, sections 2 and 3 discuss impact analysis and MSR respectively, section 4 compares IA and MSR, section 6 presents the proposed infrastructure to facilitate the comparison, and finally conclusions and future work are presented in section 7.

2. Impact Analysis

According to Arnold and Bohner [2] software-change impact analysis (a.k.a. impact analysis) is defined as the determination of potential effects to a subject system resulting from a proposed software change. The premise of impact analysis is that a proposed change may result in undesirable *side effects* and/or *ripple effects*. A side effect is a condition that leads the software to a state that is erroneous or violates the original assumptions/semantics as a result of a proposed change.

A ripple effect is a phenomenon that affects other parts of a system on account of a proposed change. The task of impact analysis is to estimate the (complete closure of) ripple effects and prevent side effects of a proposed change.

Clearly, impact analysis is a "before fact" activity. It occurs before a proposed change is actually realized. The specification of a proposed change can be in forms ranging from a high-level requirement (e.g., a textual description or formal specification) to a low-level source-code change specification. The change specification is provided in the context of artifacts such as requirement documents, design documents, source code, and test suit. The impact analysis activity starts with the change specification of a proposed change, analyzes the software, and produces a list of items that needs to be addressed as a part of a change process. Such lists are referred to as *impact* or *impact domain*. The impact obtained by examining the initial change specification of a proposed change is known as *starting impact set* (SIS). The complete-closure analysis of the items in the SIS is known as *candidate impact set* (CIS). These impact sets are (ideally speaking) validated against the actual items that are changed on account of a proposed change, known as *actual impact set* (AIS).

Dependency analysis (aka vertical analysis) and *traceability* analysis (aka horizontal analysis) are the two primary methodologies for performing impact analysis. The dependency analysis is based on the relationships between program entities (typically source-code entities such as files and functions) exhibited in various source-code based models (e.g., call-graphs, program-dependency graphs, or UML models). Broadly, dependency analysis refers to impact analysis of software artifacts at the same level of abstraction (e.g., source code to source code or design to design). Traceability analysis refers to impact analysis of software artifacts across different levels of abstractions (e.g., source code to UML). Various dependency-analysis methods based on call graphs, program slicing [10], hidden dependency analysis [4, 28, 30], lightweight static analysis approaches [22], concept analysis [27], dynamic analysis [15], hypertext systems, documentation systems, UML models [3], and Information retrieval [1] are already investigated in the literature. On the other hand, traceability analysis remains a largely ignored area in the context of impact analysis.

Impact analysis largely remains a single-version activity. That is, the underlying models used to compute the various impact sets takes into account only a single version (most typically the contemporary version) of the software system.

3. Mining Software Repositories

The term *Mining Software Repositories* (MSR) has been coined to describe a broad class of investigations into the examination of source-code versions system and other similar repositories (e.g., defect/bug tracking systems such as *Bugzilla* and *CVS*). These repositories hold a wealth of information about the actual evolution of large software systems. The premise is that empirical and systematic investigations of this (large amount of) data will shed new light on the process of software evolution and the types of changes that occur over time.

The use of software repositories such as those for source-code versions control has always been advocated as a fundamental software engineering practice. The use of software repositories and the tools that manage them facilitate sustained evolution of large software in a collaborative development/maintenance environment. However, researchers have utilized the information recorded in the software repositories in more unprecedented ways. Historically, there have been a number of efforts to examine long-term software project data to better understand software evolution. Lehman et al [16-20, 25] reported various results on the software change and nature of software evolution between 1969 and 2001 based on long-term studies of several products. The most notable results of these types of studies are the laws of software evolution [16, 17, 20], metrics of software evolution [25], classification of programs [19], and a theory of software evolution [18]. Eick et al [8] observe the phenomenon of code decay (i.e., changes to a system become difficult in terms of cost, time, and quality over its lifetime) by leveraging the software repositories.

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

A wide variety of software repositories of open-source projects are available to the public and in plenty. Researchers have realized the potential for exploring the invaluable historical data stored in the software repositories to reveal the "secrets" of various aspects of successful software evolution (e.g., source-change changes, defects, reuse, and refactorings). The eventual common goal is to learn from past failures and repeat (build on) past successes to improve the software

development and evolution processes. In summary, software repositories bring forward a new dimension of historical context in the development of future software engineering and evolution tools.

The source-code versions repositories are typically managed by tools such as *CVS* (Concurrent Versions System) and *Subversion*. They include not only change history (i.e., such as the results of *diff*) but often also include metadata about the changes (e.g., how, why, who made the changes). Researchers have devised and experimented with a variety of approaches to extract pertinent information and uncover relationships and trends in the context of software evolution (including software changes) from software repositories. This activity is very analogous (but not limited) to the field of Data Mining and Knowledge Discovery in Databases (KDD), hence the term MSR.

4. IA versus MSR

We first discuss the expressiveness and effectiveness of impact analysis. Dependency analysis of source code is typically performed using static and dynamic program analysis. Call-graph analysis is one such commonly used technique. In performing impact analysis with call graphs, the impact of a change in a function is determined to be a transitive closure of all its callers and callees. Therefore, the estimated impact set is composed of functions and typically does not provide any additional context (e.g., the precise *if* statement in a function's body that is likely to change). A number of call-graph extractors from source code are currently available, for example refer to [23]. However, call-graph analysis typically estimates impact sets that have low precision (i.e., false candidates that do not change). Moreover, it also fails to estimate some entities that actually do change (i.e., recall is typically incomplete) due to the coarse level of analysis restricted to function calls.

Static and program slicing provides granularity at the statement and variable level. Static-program slicing are typically based on the data and control flow graphs that are computationally expensive to process and analyze. It is argued in [15] that static-program slicing though providing near complete recall is likely to produce false positives and is overly conservative (i.e., accounts for program behaviors that are unlikely to be actually executed). Dynamic analysis such as dynamic-program slicing and call-path analysis [15] improve on the conservative behavior of static-program slicing. However, they are subject to the risk of lower precision and lower recall as the analysis is dependent on the executed cases. Moreover, dynamic analysis requires the additional cost of instrumentation and may not be always feasible due to the state of code during evolution (e.g., incomplete code, missing include files, etc).

Now we discuss the expressiveness and effectiveness of MSR. Version control tools identify and express changes in terms of physical attributes most typically as file and line numbers. Recently, researchers have proposed approaches that expand MSR to a more source code "aware" level (i.e., syntax and semantic) but only to a very limited degree. However, such efforts clearly demonstrate the potential of achieving much better results than working at a purely physical level. Current methods operate at a very coarse-grain granularity of source code (e.g., changes to a function or file) for mining the information of interest.

We now give a few examples of MSR approaches in the context of software-change prediction. These examples are by no means exhaustive but do represent a spectrum of different approaches. Zimmerman et al [31] used *CVS* logs for detecting evolutionary coupling between source-code entities (i.e., files, classes, methods, and variables). They employed sliding window heuristics to estimate the atomic commits (change-sets). Association-rules based on itemset mining were formed from the change-sets and used for change-prediction. Yang et al [29] used a similar technique for identifying files that frequently change together. Gall et al [9] used window-based heuristics on *CVS* logs for uncovering logical couplings and change patterns, and German et al [11] for studying characteristics of different types of changes. Hassan et al [12] analyzed *CVS* logs for software-change prediction. Source-code repositories contain differences between versions of source code. Therefore, MSR can be performed by analyzing the actual source-code differences. Such an approach is taken by the tool *Dex,* presented by Raghavan et al [24], for detecting syntactic and semantic changes from a version history of C code. In an approach by Collard et al [5, 21] a syntactic-differencing approach called *meta-differencing* is introduced. The approach allows you to ask syntax-specific questions about differences (e.g., was a method *A* added?).

The historical context (i.e., real impact sets) given by MSR can be utilized to assess the quality of the impact sets produced by impact analysis techniques. Additionally, the historical context can be utilized to augment the impact analysis models to improve their change prediction power, if that is the case. Similarly, software entities that are not predicted to change by MSR but are predicted correctly by impact analysis could be used to validate MSR. Therefore, impact analysis and MSR could be used to cross validate each other and improve software-change prediction tools.

5. Research Program

Our on-going research directly addresses the issue of source-code aware mining of software repositories. To

this end, we refer source-code aware MSR as approaches that take into account finer granularity of source-code differences and other source-code models. Our basic research interest is in examining the impact on the MSR approaches with regards to the fine-grain granularity differences and analysis.

The MSR approaches proposed for discovering source-code entities (e.g., functions) change dependencies or trends are typically based on version repositories metadata analysis. For example, entities are considered to have change dependency, if they are changed together in the same commit operation. It is inherently assumed that such dependencies are "hidden" relations that are not explicitly documented or could be left uncovered by the analysis of a single (current) version. However, to our knowledge no research investigation has been conducted to validate that this is truly the case and if/how these so-called "hidden" dependencies correspond to the relationships present in the various source code models (e.g., call-graphs, UML class models, etc).

We refer to this problem as identification of pure-evolutionary dependencies. We are concerned with what are the change-prone dependencies between source-code entities that are exclusively revealed by mining historical information (i.e., "hidden dependencies") and not by any source-code or high-level abstractions models?

Our hypothesis is that the source-code aware MSR approaches directly contribute to improved tool support for software evolution. In order to study the above stated problems and validate our hypothesis within the scope of this work, we specifically investigate the following research questions with regards to software-change analysis:

- Do fine-grain source-code differences lead to better support for impact analysis in terms of precision, recall, location, and context of software changes?
- Can a correspondence be made between "hidden" dependencies and source-code models (e.g., call-graphs, UML class models)?
- Which kinds of "hidden" dependencies can or can't be represented by relations in the source-code models, if a correspondence between them can be drawn? Which source-code models are "good/bad" filters of false hidden dependencies?
- How to utilize the correspondence (if any) between "hidden" dependencies and relations in the source-code models to identify evolutionary (i.e., change) dependencies for impact analysis?

Investigations by Zimmermann et al have partially shown the benefits of further processing the information directly available from source-code repositories for change prediction. In their study in [31], there was no significant difference in precision and recall values between file and syntactic entities (i.e., classes, methods, and variables) with respect to change prediction tasks. However, we argue that there is an implicit gain in terms of the context (i.e., the exact location of a predicted entity that needs to be changed) available to the maintainer. For example, predicting a change at the syntactic-entity level rather than the file level simplifies the manual effort as it is only needed to examine the predicted entities and not the whole file. This leads to the issue of extending MSR in terms of the *source-code awareness* with the added cost of fine-granularity processing.

The goal of MSR is uncover the past successes (failures) from the historic information and repeat or better (avoid) the evolution of the software system 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 may be subject to threat of irrelevant information. The design or operational assumptions of the system may no longer be the same 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 the 1001^{th} version. A series of changes from version 50 to version 200 could be attributed to an unstable unit in the system that has now stabilized. Once again, the information about these changes may be irrelevant for future change predictions.

On the other hand considering too few past versions from the current state of the system imposes the risk of missing important relevant information. For example, a current version of a system may be in the middle of a refactoring that is achieved by a sequence of changes (versions). At least the past versions to the staring point from where the refactoring started are needed to first confirm the kind of refactoring taking place and predict the remaining steps to complete it. In summary, the question of historical information to consider reduces to how much and which portion of history to mine such that it is not too much to include irrelevant and not too little to miss important relevant information. The answer to this question is one of the prime factors that could affect the effectiveness of MSR techniques.

6. Proposed Infrastructure

In order to support the investigation of the source-code awareness problem, the *srcML* [7] and *srcDiff* [21] infrastructure will be utilized and further developed. *srcML* is an *XML* representation of source code that

explicitly embeds the syntactic structure inherently present in source-code text with *XML* tags. The format preserves all the original source code contents including comments, white space, and preprocessor directives. *srcDiff* is an extension of *srcML* format to further embed syntactic difference information between source-code documents. Currently, *srcDiff* representation consists of two versions of a source-code document merged together along with the overlaid difference information between them. Source code is represented in *srcML*. The line-based differences typically produced by *diff* like tools are mapped to corresponding syntactic units.

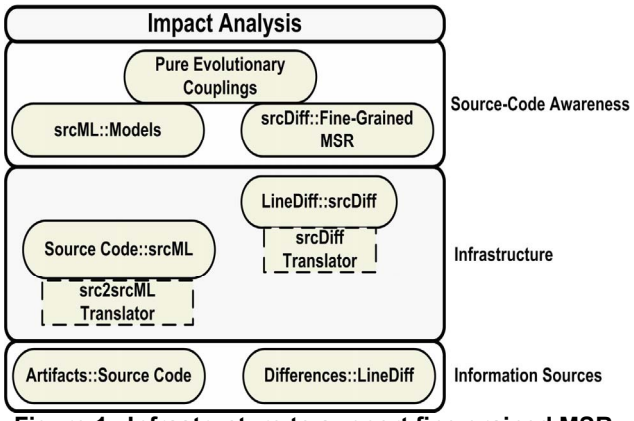

Figure 1. Infrastructure to support fine-grained MSR and investigation of pure-evolutionary dependencies

The schematic of the proposed solution is presented in Figure 1. The source-code artifacts and their line-based differences directly available from a source-code repository will be transformed to *srcML* and *srcDiff* representations. The tools *srcML* translator, namely *src2srcML* and *srcDiff* translator will be utilized to achieve the above transformations. A preliminary version of the *src2srcML* [13] was evaluated on the *CppETS* [26] benchmark for the fact extraction tasks [6]. A highly extended, robust, and efficient *srcML* translator is currently available at www.sdml.info. This mature tool is used on multiple research projects. Also, a proof-of-concept *srcDiff* translator resulted from the work on *Meta-Differencing* [5]. The *srcDiff* format and *srcDiff* translator will be further developed to meet the needs of the proposed research as and when they are identified. Also, the capability and features of *srcML* representation will be used to easily extract and derive various abstraction models such as call-graphs, program dependency graphs, and UML Class models from source code. These models will be utilized to perform impact analysis in the traditional sense (i.e., analyzing a single version). The *srcDiff* representation allows to perform queries on fine-grained differencing, therefore, allowing software-change prediction based on history at various syntactic levels (i.e., fine-grained MSR). Impact analysis on the models derived from srcML and the fine-grained MSR facilitated by srcDiff supports the detection of pure-evolutionary dependencies (i.e., exclusive to *MSR*). The pure-evolutionary dependencies (if exists) will be augmented to the single-version models to extend the scope of impact analysis. In summary, the synergy of *srcML*, *srcDiff*, and the "standard" tool-support (i.e., *XML* processing tools) forms a solid base to address the fine-grained MSR and pure-evolutionary couplings research questions.

7. Conclusions and Future Work

The investigation of fine-grained MSR and identification of pure-evolutionary dependencies remains an interesting and important problem to realize the true value of MSR. This investigation will serve as a basis for validating the tradeoff between the additional mining cost of historical information (i.e., actual-impact sets) and the improved effectiveness for software-change prediction (i.e., more accurate estimated-impact sets). Currently we are developing a generic tool that will facilitate querying on the reverse-engineered models from source code. We have also developed a tool based on data-mining technique that analysis data in software-repositories and produces rules that could be used for software-change prediction. These rules facilitate prediction of a sequence of entities that are likely to change in a specific (partial) order [14]. We plan to evaluate the effectiveness of estimated and actual impact sets on a number of open-source projects (e.g., *KDE* and *Apache*).

8. References

[1] Antoniol, G., Canfora, G., Casazza, G., and Lucia, A., "Identifying the Starting Impact Set of a Maintenance Request: A Case Study", in Proceedings of Conference on Software Maintenance and Reengineering, Zurich, Switzerland, February 29-March 03 2000, pp. 227-231.

[2] Arnold, R. and Bohner, S., Software Change Impact Analysis, Wiley, 1996.

[3] Briand, L., Labiche, Y., and Soccar, G., "Automating Impact Analysis and Regression Test Selection Based on UML Designs", in Proceedings of International Conference on Software Maintenance (ICSM'02), Montreal, Quebec, Canada, October 03-06 2002, pp. 252-261.

[4] Chen, K. and Rajlich, V., "RIPPLES: Tool for Change in Legacy Software", in Proceedings of International Conference on Software Maintenance (ICSM'01), Florence, Italy, November 07-09 2001, pp. 230-239.

[5] Collard, M. L., Meta-Differencing: An Infrastructure for Source Code Difference Analysis, Kent State University, Kent, Ohio USA, Ph.D. Dissertation, 2004.

[6] Collard, M. L., Kagdi, H. H., and Maletic, J. I., "An XML-Based Lightweight C++ Fact Extractor", in Proceedings of 11th IEEE International Workshop on Program Comprehension (IWPC'03), Portland, OR, May 10-11 2003, pp. 134-143.

[7] Collard, M. L., Maletic, J. I., and Marcus, A., "Supporting Document and Data Views of Source Code", in Proceedings of ACM Symposium on Document Engineering (DocEng'02), McLean VA, November 8-9 2002, pp. 34-41.

[8] Eick, S. G., Graves, T. L., Karr, A. F., Marron, J. S., and Mockus, A., "Does Code Decay? Assessing the Evidence from Change Management Data", IEEE Transactions on Software Engineering (TSE), vol. 27, no. 1, 2001, pp. 1-12.

[9] Gall, H., Hajek, K., and Jazayeri, M., "Detection of Logical Coupling based on Product Release History", in Proceedings of International Conference on Software Maintenance (ICSM'98), 1998, pp. 190-199.

[10] Gallagher, K. and Lyle, J., "Using Program Slicing in Software Maintenance", Transactions on Software Engineering, vol. 17, no. 8, August 1991, pp. 751-762.

[11] German, D. M., "An Empirical Study of Fine-Grained Software Modifications", in Proceedings of 20th IEEE International Conference on Software Maintenance (ICSM'04), 2004, pp. 316-25.

[12] Hassan, A. E. and Holt, R. C., "Predicting Change Propagation in Software Systems", in Proceedings of 20th IEEE International Conference on Software Maintenance (ICSM'04), 2004, pp. 284-93.

[13] Kagdi, H., Using An Island Grammar Approach for Lightweight Parsing: A C++ To SrcML Translator, Kent State University, Kent, Masters Thesis, 2003.

[14] Kagdi, H., Yusuf, S., and Maletic, J. I., "Mining Sequences of Changed-files from Version Histories", in Proceedings of International Workshop on Mining Software Repositories (MSR'06), Shanghai, China, May 22-23, 2006, pp. 47-53.

[15] Law, J. and Rothermel, G., "Whole Program Path-Based Dynamic Impact Analysis", in Proceedings of 25th International Conference on Software Engineering, Portland, Oregon, May 03 -10 2003, pp. 308-318.

[16] Lehman, M., "On Understanding Laws, Evolution and Conservation in the Large Program Life Cycle", Journal of Systems and Software, vol. 1, no. 3, 1980, pp. 213-221.

[17] Lehman, M., Perry, D., and Ramil, J. F., "On Evidence Supporting the FEAST Hypothesis and the Laws of Software Evolution", in Proceedings of Fifth International Symposium on Software Metrics (METRICS98), Bethesda, Maryland, Nov. 20-21 1998, pp. 84-88.

[18] Lehman, M. and Ramil, J. F., "An Approach to a Theory of Software Evolution", in Proceedings of International Workshop on Principles of Software Evolution (IWPSE), Vienna, Austria, September 10 - 11 2001, pp. 70-74.

[19] Lehman, M. and Ramil, J. F., "Evolution in Software and Related Areas", in Proceedings of International Workshop on Principles of Software Evolution (IWPSE), Vienna, Austria, September 10 - 11 2001, pp. 1-16.

[20] Lehman, M. M. and Belady, L. A., Program evolution: processes of software change, Academic Press Professional, Inc., 1985.

[21] Maletic, J. I. and Collard, M. L., "Supporting Source Code Difference Analysis", in Proceedings of IEEE International Conference on Software Maintenance (ICSM'04), Chicago, Illinois, September 11-17 2004, pp. 210-219.

[22] Moonen, L., "Lightweight Impact Analysis using Island Grammars", in Proceedings of 10th International Workshop on Program Comprehension (IWPC'02), Paris, France, June 27-29 2002, pp. 219-228.

[23] Murphy, G. C., Notkin, D., Griswold, W. G., and Lan, E. S., "An Empirical Study of Static Call Graph Extractors", ACM Transactions on Software Engineering and Methodology, vol. 7, no. 2, April 1998, pp. 158-191.

[24] Raghavan, S., Rohana, R., Podgurski, A., and Augustine, V., "Dex: A Semantic-Graph Differencing Tool for Studying Changes in Large Code Bases", in Proceedings of 20th IEEE International Conference on Software Maintenance (ICSM'04), Chicago, Illinois, September 11 - 14 2004, pp. 188-197.

[25] Ramil, J. F. and Lehman, M., "Metrics of Software Evolution as Effort Predictors - A Case Study", in Proceedings of Proceedings of the International Conference on Software Maintenance (ICSM00), San Jose, California, USA, October 11-14 2000, pp. 163-172.

[26] Sim, S. E., Holt, R. C., and Easterbrook, S., "On Using a Benchmark to Evaluate C++ Extractors", in Proceedings of 10th International Workshop on Program Comprehension, Paris, France, 2002, pp. 114-123.

[27] Tonella, P., "Using a Concept Lattice of Decomposition Slices for Program Understanding and Impact Analysis", Transactions on Software Engineering, vol. 29, no. 6, June 2003, pp. 495-509.

[28] Vaclav, R., "A Model for Change Propagation Based on Graph Rewriting", in Proceedings of International Conference on Software Maintenance (ICSM '97), Bari, ITALY, October 01-03 1997, pp. 84-91.

[29] Ying, A. T. T., Murphy, G. C., Ng, R., and Chu-Carroll, M. C., "Predicting Source Code Changes by Mining Change History", IEEE Transactions on Software Engineering, vol. 30, no. 9, September 2004, pp. 574 - 586.

[30] Yu, Z. and Rajlich, V., "Hidden Dependencies in Program Comprehension and Change Propagation", in Proceedings of Ninth International Workshop on Program Comprehension (IWPC'01), Toronto, Canada, May 12-13 2001, pp. 293-299.

[31] Zimmermann, T., Zeller, A., Weissgerber, P., and Diehl, S., "Mining Version Histories to Guide Software Changes", IEEE Transactions on Software Engineering, vol. 31, no. 6, 2005, pp. 429-445.