Combining Single-Version and Evolutionary Dependencies for Software-Change Prediction

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

Abstract

The paper advocates the need for the investigation and development of a software-change prediction methodology that combines the change sets estimated from software dependency analysis (via single-version analysis) and the actual change sets found in software version histories (via multiple-version analysis). Traditionally prescribed methodologies such as Impact Analysis (IA) are based on the former, whereas a more recent methodology, Mining Software Repository (MSR), is based on the latter. The research hypothesis is that combining these two methodologies will result in an overall improved support for software-change prediction.

1. Introduction

Changes to a software system occur due to various causes such as bug correction, feature addition, and design improvement. Irrespective of their underlying causes, changes may not always be localized to a single element. Such changes can lead to potential side effects and/or violations of the underlying assumptions. Software-change prediction methodologies that provide all the entities that need to be appropriately co-changed are important for sustained evolution of a software system [5, 14]. Two broad groups of methodologies are described in the literature for supporting software changes. The approaches described under the area of *Software-Change Impact Analysis (IA)* are among the early efforts that support change estimation [1]. *Mining Software Repositories (MSR)* is a growing area of research that has shown the emergence of approaches for supporting change predictions [6-8, 18].

Bohner and Arnold surveyed IA methodologies in 1996 [1], and a number of approaches based on improved static and dynamic analyses are proposed thereafter (e.g., [2, 3, 13, 16, 17]). An extensive examination of the MSR approaches was recently completed [9], of which a preliminary survey of six approaches that support software changes is discussed in [10]. These works indicate that IA and MSR methodologies can undertake orthogonal perspectives towards meeting the common goal of supporting change management.

Another observation is that both IA and MSR methodologies can be improved in their *expressiveness* and *effectiveness* for supporting software-change prediction [11]. The expressiveness of a methodology is defined as the granularity of software artifacts (e.g., file, class, and function) and the change context (e.g., a parameter added to a function, a member variable deleted from a class) at which a prediction is made. The effectiveness of a methodology is described in terms of accuracy (i.e., precision and recall) along with the computational cost.

Our goal is to examine whether the combined use of IA and MSR approaches results in an improved expressiveness and effectiveness for software-change prediction[1]. We feel that the combination of these approaches will result in more accurate results. An empirical investigation is currently being conducted on a number of open-source systems (e.g., *KDE* and *Apache*) to evaluate a hybrid approach that combines the two. Also, a number of tools to support mining and analysis on a fine-grained level are being developed for supporting this investigation. In this discussion, the effectiveness of a hybrid approach is of principal focus.

The remainder of the paper is organized as follows. Section 2 briefly describes IA and MSR. Section 3 discusses the developed infrastructure for a combined approach. Section 4 describes the empirical validation framework. Finally, Section 5 presents conclusions.

2. IA and MSR

The term *Dependency Analysis* is used to refer to impact analysis of software artifacts at the same level of abstraction (e.g., source code to source code and design to design) [1]. The basic premise of a typical dependency-analysis approach is to use the relationships between entities (e.g., files and functions) in an

[1] Here, the focus is on the source code change prediction, however the examination remains of interest for all software artifacts.

abstraction model (e.g., call-graphs, program-dependency graphs, or UML models), and/or dynamic behavior (e.g., run-time profiling data), of a single snapshot of a program (e.g., program version/release). A relationship between entities in a model is considered as an indicator of a change dependency between them. That is, if an entity is changed, the "related" entities are estimated to change.

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

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

One approach in MSR is to analyze commits (i.e., a set of changed artifacts checked-in together) and metadata in software repositories to infer *evolutionary dependencies* or *co-changes* between artifacts [18]. Since software repositories typically provide differences only at file and line number, the expressiveness of the entities involved in a co-change is dependent on the further fine-grain analysis performed (e.g., to achieve a syntactic and/or semantic level of granularity). A unique advantage of such a MSR approach is that only changes to entities are analyzed compared to complete analyses of all the artifacts. However, on the other end, such an approach may fail to predict "unseen" changes in the past, and may incorrectly predict obsolete changes to entities that do not exist anymore.

A straightforward step is to combine the two somewhat different approaches of dependency analysis and MSR[2] approach. Using both the software dependencies and evolutionary dependencies could help improve the overall change prediction methodology. The actual changes in a software repository can be utilized to assess the quality of the impact sets produced by impact analysis techniques. Additionally, the historical context can be utilized to augment the impact analysis models to improve their change prediction power. Similarly, software entities that are not predicted to change by MSR but are predicted correctly by impact analysis could be used to validate MSR. Therefore, impact analysis and MSR could be used to cross validate, refine, and supplement each other.

[2] MSR is much broader than supporting software changes. Here, for brevity we limit it to approaches for change analysis and prediction.

3. Supporting a Combined Approach

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

The abstraction models which are the underlying basis of dependency analysis will be constructed from the *srcML* representation [4, 15]. *srcML* is an *XML* representation of source code that explicitly embeds the syntactic structure inherently present in source code text with *XML* tags. The format preserves all the original source code contents including comments, white space, and preprocessor directives. The capability and features of *srcML* representation can be used to easily extract facts [4] with standard XML processing tools, and derive abstraction models such as call-graphs, program-dependency graphs, and UML models from source code.

A frequent-pattern mining tool, namely *sqminer* is developed to uncover co-changes from commits stored in a source code repository. *sqminer* was applied to mine co-changes at the file level [12]. For example, sequences of changed-files (i.e., co-changes) such as {f1}→{f2} and {f4}→{f5} were mined. The symbol → in the sequence {f1}→{f2} indicates that changes in {f1} *happened before* {f2}. A differencing tool, namely *codeDiff* is developed based on *srcML* and a word differencing tool, namely *dwdiff*. It takes two versions of a source code file and produces differences between them at a syntactic level. *codeDiff* will be used to process the differences in a source code file of a commit to fine-grained syntactic level. This will help mining co-changes with *sqminer* at fine-grain syntactic levels.

4. Evaluation Framework

The first part of the evaluation is to select the abstraction model for dependency analysis and the mining methodology for MSR. For dependency analysis, the estimated changes will be based on the abstraction models formed by static analysis (e.g., static call graphs and program-dependency graphs) and dynamic analysis

(dynamic call graphs and profiling). The granularity of entities predicted from both approaches will be appropriately matched. For example, if the expressiveness of estimated entities is functions/methods and variables from dependency analysis, then for MSR the co-changed will be mined at the same granularity.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

A combined approach for change prediction that uses only the intersection of estimations of dependency analysis and estimations of co-change approach is termed as the *Conjunctive Approach.* Precision and recall are:

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

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

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

The exclusive dependency-analysis estimation set, D_i - M_i that has a high precision and recall represents change dependencies that could only be correctly predicted by the dependency-analysis approach. Therefore, indicating

that co-change mining approach alone may be insufficient. Similarly, very the low-accuracy exclusive sets M_i - D_i and D_i - M_i may indicate when not to use co-change approach and dependency analysis respectively.

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

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

5. Conclusions

The contributions of this investigation are a step towards answering our overarching research question as to what are the exclusive and potentially synergistic benefits of IA and MSR methodologies with regards to change prediction. The proposed evaluation will provide an empirical basis to help answer this question and provide a recommendation system for different classes of changes. We believe that identification of pure-evolutionary dependency is an interesting and important problem to appreciate the true value of MSR approaches. Another issue that will be addressed (but not discussed here) is the co-relation between fine-grained expressiveness and effectiveness. That is, does predicting changes at a finer granularity with context improve accuracy/cost?

6. References

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

[2] Briand, L., Labiche, Y., and Sullivan, L., "Impact Analysis and Change Management of UML Models", in Proceedings International Conference on Software Maintenance (ICSM'03), Amsterdam, The Netherlands, Sept. 22-26 2003, pp. 256-265.

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

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

[5] 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", Trans on Software Engineering, 27, 1, 2001, pp. 1-12.

[6] Gall, H., Jazayeri, M., and Krajewski, J., "CVS Release History Data for Detecting Logical Couplings", in Proceedings Workshop on Principles of Soft Evolution, 2003, pp. 13-23.

[7] German, D. M., "An Empirical Study of Fine-Grained Software Modifications", in Proceedings 20th IEEE International Conference on Software Maintenance (ICSM'04), Chicago, Illinois, September, 11-17 2004, pp. 316-325.

[8] Hassan, A. E. and Holt, R. C., "Predicting Change Propagation in Software Systems", in Proceedings 20th IEEE International Conference on Software Maintenance (ICSM'04), Chicago, Illinios, September, 11-17 2004, pp. 284-293.

[9] Kagdi, H., Collard, M., and Maletic , J. I., "A Survey and Taxonomy of Approaches for Mining Software Repositories in the Context of Software Evolution", *Journal of Software Maintenance and Evolution: Research and Practice*, submitted manuscript, 2006

[10] Kagdi, H., Collard, M. L., and Maletic , J. I., "Towards a Taxonomy of Approaches for Mining of Source Code Repositories", in Proceedings 2nd International Workshop on Mining Software Repositories (MSR'05), St. Louis, Missouri 2005 pp. 90-94.

[11] Kagdi, H. and Maletic , J. I., "Software-Change Prediction: Estimated+Actual ", in Proceedings 2nd International IEEE Workshop on Software Evolvability (SE'06), Philadelphia, PA, September, 24 2006, pp. 38-43.

[12] Kagdi, H., Yusuf, S., and Maletic, J. I., "Mining Sequences of Changed-files from Version Histories", in Proceedings 3rd International Workshop on Mining Software Repositories (MSR'06) Shanghai, China, May 22-23, 2006 2006, pp. 47-53.

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

[14] Lehman, M., "On Understanding Laws, Evolution and Conservation in the Large Program Life Cycle", Journal of Systems and Software, 1, 3, 1980, pp. 213-221.

[15] Maletic, J. I., Collard, M. L., and Marcus, A., "Source Code Files as Structured Documents", in Proceedings 10th IEEE International Workshop on Program Comprehension (IWPC'02), Paris, France, June 27-29 2002, pp. 289-292.

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

[17] Tonella, P., "Using a Concept Lattice of Decomposition Slices for Program Understanding and Impact Analysis", Trans. on Software Engineering, 29, 6, June 2003 2003, pp. 495-509.

[18] Zimmermann, T., Zeller, A., Weissgerber, P., and Diehl, S., "Mining Version Histories to Guide Software Changes", Trans. on Soft. Engineering, 31, 6, 2005, pp. 429-445.