Managing Software Evolution through Semantic History Slicing

Yi Li
Department of Computer Science
University of Toronto
Toronto, ON, Canada
liyi@cs.toronto.edu

Abstract—**Software change histories are results of incremental updates made by developers. As a side-effect of the software development process, version history is a surprisingly useful source of information for understanding, maintaining and reusing software. However, traditional commit-based sequential organization of version histories lacks semantic structure and thus are insufficient for many development tasks that require high-level, semantic understanding of program functionality, such as locating feature implementations and porting hot fixes. In this work, we propose to use well-organized unit tests as identifiers for corresponding software functionalities. We then present a family of automated techniques which analyze the semantics of historical changes and assist developers in many everyday practical settings. For validation, we evaluate our approaches on a benchmark of developer-annotated version history instances obtained from real-world open source software projects on GitHub.**

Index Terms—**software changes; version histories; program analysis; software reuse.**

I. Problem

Software Configuration Management systems (SCMs) are widely used in software development practices. These systems, e.g., Git [1] and SVN [2], are useful for capturing incremental changes made by developers, examining or reverting changes, identifying developers responsible for a specific change, creating development streams, and more. Incremental changes are manually grouped by developers to form *commits* (a.k.a. *change sets*). Commits are stored sequentially and ordered by their time stamps, so that it is convenient to trace back to any version in the history. *Branching* is another construct provided by most modern SCM systems. Branches are used, for example, to store a still-in-development prototype version of a project or to store multiple project variants targeting different customers.

However, the sequential organization of changes is inflexible and lacks support for many tasks that require high-level, semantic understanding of program functionality [3], [4]. For example, developers often need to locate and transfer functionality from one branch to another: either for porting bug fixes or for splitting large chunk commits into multiple functionally-independent pull requests. Identifying failure-inducing changes in version histories is another challenge that developers face in their work.

Several SCM systems provide mechanism of "replaying" commits on a different branch, e.g., the `cherry-pick` command in Git. Yet, little support is provided for matching high-level functionality with commits that implement it: SCM systems only keep track of temporal and text-level dependencies between the managed commits. The job of identifying the exact set of commits implementing the functionality of interest is left to the developers.

Motivated by these challenges, we propose a new semantics-based view of software version histories, where a set of related changes satisfying a common high level property (a.k.a. slicing criteria) is known as a *semantic history slice*. As one concrete instantiation, test cases exercising a software functionality can be used as slicing criteria to identify the changes implementing the particular functionality. This dissertation hypothesizes that the proposed organization of version histories is effective in software evolution tasks including software understanding, maintenance, and reuse.

II. Related Work

The proposed semantic history slicing problem is most related to change impact analysis and change history analysis including history understanding and manipulation.

Change Impact Analysis. *Change Impact Analysis* [5] solves the problem of determining the effects of source code modifications. It usually means selecting a subset of tests from a regression test suite that might be affected by the given change, or, given a test failure, deciding which changes might be causing it.

Research on impact analysis can be roughly divided into three categories: the *static* [6], [5], *dynamic* [7] and *combined* [8], [9] approaches. The work most related is on the combined approaches to change impact analysis. Ren et al. [8] introduced a tool, Chianti, for change impact analysis of Java programs. Chianti takes two versions of a Java program and a set of tests as the input. First, it builds dynamic call graphs for both versions before and after the changes through test execution. Then it compares the classified changes with the old call graph to predict the affected tests; and it uses the new call graph to select the affecting changes that might cause the test failures. FaultTracer [9] improved Chianti by extending the standard dynamic call graph with field access information. Similar techniques can be used to identify changes relating to a slicing criterion. However, another challenge in our problem is to process and analyze the identified changes and ensure that the final results are semantics-preserving and well-formed.

History Understanding and Manipulation. There is a large body of work on analyzing and understanding software histories. The basic research goals are retrieving useful information from change histories to help understand development practices [4], [10], [11], [12], localize bugs [13], [14], and support predictions [15], [16].

The most relevant take on history analysis is to create flexible views of the change histories at various granularities instead of using the fixed commit-based representation. Some notable approaches include *history slicing* [12] and *history transformation* [17], [4]. The promise of these techniques is to provide users the most convenient and effective ways of interacting with change histories and better facilitate the specific software evolution tasks at hand. For example, Muşlu et al. [4] introduced the concept of semantics summarization view which clusters original sequence of commits into semantically related high-level logical groups. Our proposed approach can be viewed as an implementation of this concept.

Delta debugging [13] uses divide-and-conquer-style iterative test executions to narrow down the causes of software failures. It has been applied to minimize the set of changes which cause regression test failures. This problem can be considered as finding minimal semantic history slices with respect to the failure-inducing properties.

III. Preliminaries

Functionality Tests. We assume that high level software functionalities such as features and bug fixes can be captured by tests and the execution trace of a test is deterministic [18]. A *test* t is a predicate $t : P \mapsto \mathbb{B}$ such that for a given program p, $t(p)$ is true if the test succeeds, and false otherwise. A *test suite* is a collection of tests that can exercise and demonstrate the functionality of interest. Let a test suite T be a set of test cases $\{t_i\}$. We write $p \models T$ if and only if a program p passes all tests in T, i.e., $\forall t \in T \cdot t(p)$.

Commit and Commit History. Let a *commit* be a partial function $\Delta : P \nrightarrow P$ which takes a program version p and transforms it to produce a new program version $\Delta(p)$. A commit is a collection of *hunks* [19], [20] $\{\delta_0, \ldots, \delta_n\}$, in no particular order, each representing a set of line changes with an approximate locality. Composing hunks is equivalent to applying the original commit, i.e., $\Delta = \delta_0 \circ \cdots \circ \delta_n$.

A *commit history* is a sequence of commits $H = \langle \Delta_1, \ldots, \Delta_k \rangle$. A *sub-history* is a sub-sequence of a history, i.e., a sequence derived by removing changes from H without altering the ordering. We write $H' \subseteq H$ indicating that H' is a sub-history of H, and refer to $\langle \Delta_i, \ldots, \Delta_j \rangle$ as $H_{i..j}$. We use $SH(H)$ to denote the set of all sub-histories of H.

IV. Proposed Solutions

To tackle the problem described in Sect. I, we propose a formal definition of *semantics-preserving slice* and then discuss two different approaches for finding such history slices.

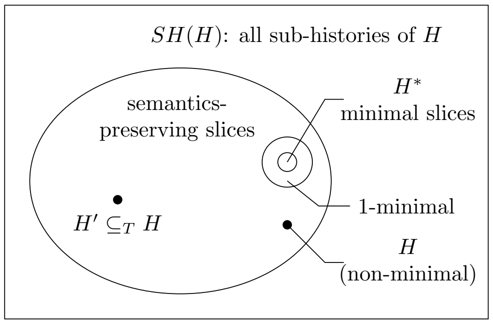

Fig. 1. Relationships between various history slices.

A. Semantics-Preserving History Slices

Consider a program $p_0 \in P$ and its n subsequent versions $p_1, \ldots, p_n$ such that they are all well-formed. Let H be the original commit history from p_0 to p_n, i.e., $H_{1..i}(p_0) = p_i$ for all integers $0 \leq i \leq n$. Let T be a set of tests passed by p_n, i.e., $p_n \models T$; T is fixed once chosen.

Definition 1. *(Semantics-preserving slice [20]). A* semantics-preserving slice *of history H with respect to T, denoted by $H' \lhd_T H$, is a sub-history of H, i.e., $H' \subseteq H$, such that $H'(p_0) \models T$.*

Definition 2. *(Minimal semantics-preserving slice [21]). A semantics-preserving slice H^* is a* minimal *if $\forall H_{sub} \subset H^* \cdot H_{sub} \not\models T$.*

As shown in Fig. 1, there are several special kinds of semantics-preserving slices. First, H is a semantics-preserving slice of itself, but it may not be *minimal*. Second, minimal semantic slices (H^*) are slices which are semantics-preserving and cannot be reduced further. Finally, computing minimal semantics-preserving slices is expensive [21], so we often compute an approximation known as the *1-minimal* semantic slice – a slice which cannot be further reduced by removing *any single commit*. In practice, 1-minimal slices are often minimal [22].

B. Finding Semantics-Preserving Slices

With the presence of adequate tests for a functionality, and the corresponding development history, *semantic history slicing* is a technique which uses tests (slicing criteria) to identify commits in the history (i.e., a semantics-preserving slice) that contribute to the implementation of the given functionality.

A trivial but uninteresting solution to this problem is the original history H itself. Shorter slicing results are preferred over longer ones, and the optimal slice is the shortest sub-history that satisfies the above properties. However, the optimality of the sliced history cannot always be guaranteed by polynomial-time algorithms. Since the test case can be arbitrary, it is not hard to see that for any program and history, there always exists a worst case input test that requires enumerating all 2^k sub-histories to find the shortest one. The naïve approach of enumerating sub-histories is not feasible as the compilation and running time of each version can be substantial. Even if a compile and test run takes just one minute, enumerating and building all sub-histories of only twenty commits would take approximately two years. In fact, it can be shown that the optimal semantic slicing problem is NP-complete by reduction from the set cover problem. We omit the details of this argument here.

To balance between performance and precision, we devise two different algorithms for semantic history slicing, namely the *static* slicing and *dynamic* slicing approaches.

Static Slicing Based on Dependency Analysis. The static approach mostly relies on static analysis of dependencies between change sets and is therefore much cheaper in terms of running time. CSLICER [20] is an efficient static slicing algorithm which requires only a one-time effort for compilation and test execution.

The actual slicing process consists of two phases, a generic history slicing algorithm which is independent of any specific SCM system in use, and an SCM adaptation component that adapts the output produced by the slicing algorithm to specifics of SCM systems. The slicing algorithm conservatively identifies all atomic changes in the given input history that contribute to the *functional* and *compilation* correctness of the functionality of interest. The SCM adaptation component then maps the collected set of atomic changes back to the commits in the original change history. It also takes care of merge conflicts that can occur when cherry-picking commits in text-based SCM systems such as SVN and Git. CSLICER is designed to be conservative in the first phase and thus can be imprecise.

Dynamic Slicing Through Delta Refinement. In contrast, the dynamic approach executes tests multiple times and directly observes the test results while attempting to shorten the history slices iteratively. The semantic slices found by the dynamic approach are guaranteed to be minimal, but the running time is usually much longer.

DEFINER [22] derives a small and precise semantic slice through the more expensive repeated test executions in a divide-and-conquer fashion that is very similar to *delta debugging* [13]. The high-level idea is to partition the input history by dropping some subset of the commits and opportunistically reduce the search space when the target tests pass on one of the partitions, until a minimal partition is reached. To speed up the process, DEFINER also uses observed test pass/fail signals and dynamic program invariants to predict the significance of change sets with respect to the target tests.

DEFINER operates on the commit-level, and the history slices produced by DEFINER is guaranteed to be *1-minimal* – removing any single commit from the history slice will break the desired semantic properties.

V. APPLICATIONS

We have successfully applied the history slicing techniques in many development tasks including back-porting bug fixes [20], creating self-contained and easy-to-merge pull requests [21], locating feature implementations and building feature models [23] to assist evolution understanding.

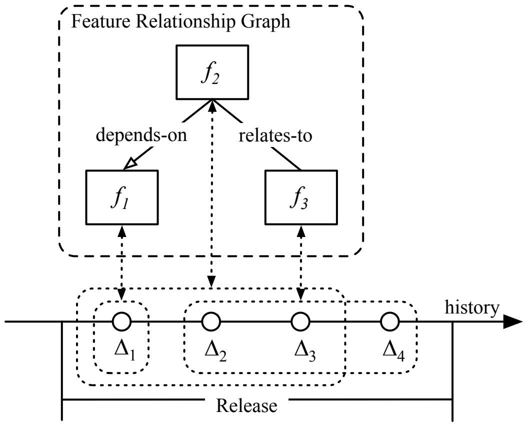

Fig. 2. Feature-implementing changes and feature relationship graph extracted from a release history.

A1. Porting Functionalities Across Versions. The first use case of semantic slicing is to identify the set of commits required for back-porting a functionality to earlier versions of a software project. Even in very disciplined projects, when such commits can be identified by browsing their associated log messages, the functionality of interest might depend on earlier commits in the same branch. To ensure correct execution of the desired functionality, all change dependencies have to be identified and migrated as well, which is a tedious and error-prone manual task. Given test cases for the functionalities to be ported, semantic slicing techniques can automatically compute the required changes and, at the same time, effectively avoid including unnecessary changes.

A2. Creating Pull Requests. Another important use case of semantic slicing is creating logically clean and easy-to-merge pull requests. Often, a developer works on multiple functionalities at the same time which could result in mixed commit histories concerning different issues. However, when submitting pull requests for review, contributors should refrain from including unrelated changes as suggested by many project contribution guidelines. Despite the efforts of keeping the development of each issue on separate branches, isolating each functional unit as a self-contained pull request is still a challenging task. For a particular pull request, the test cases created for validation can be used as slicing criteria to identify relevant commits from the developers' local histories in their forked repositories.

A3. Identifying Features and Feature Relationships. Identifying features in cloned product line variants is important for a variety of software development tasks such as sharing features between variants and refactoring cloned variants into single-copy software product line (SPL) representations. Semantic slicing is an effective way of locating feature-implementing changes in software version histories with the presence of feature tests. For example, Fig. 2 shows the mapping between features and their corresponding commits identified by semantic slicing within a single release history: $f_1 \mapsto \{\Delta_1\}, f_2 \mapsto \{\Delta_1, \Delta_2, \Delta_3\}, f_3 \mapsto \{\Delta_2, \Delta_3, \Delta_4\}$. In addition, the resulting feature relationship graph is useful for understanding dependencies and connections between features from an evolutionary view point. Each valid product has to respect the inferred *depends-on* relationships in order to function correctly. The *relates-to* relationships indicate connections between features. They often reveal underlying hidden dependencies which are essential across the system.

A4. Evolution Management Framework. In order to unify different semantic slicing algorithms and provide software developers a flexible and ready-to-use tool for various evolution management tasks, we plan to build a cloud-based history slicing service framework. The front-end of the tool chain is a Web application closely integrated with the GitHub APIs to allow access to users' repository meta data and project version histories. The user interface visualizes various options and slicing results to allow more user friendly interactions with the underlying techniques. The back-end runs on a central server and implements a number of important optimizations including parallelization and caching of slicing results. It also seamlessly switches between different history slicing algorithms according to specific usage scenarios.

VI. Plan for Evaluation

In our preliminary work, we have developed experimental support to evaluate the efficiency and effectiveness of our semantic slicing techniques. More specifically, we implemented prototype tools for both the CSlicer and the Definer slicing algorithms. The tools work with Java projects hosted in Git repositories and they are available at: bitbucket.org/liyistc/gitslice. We also constructed a dataset [24] of 100 history slicing problem instances collected from 8 real-world software projects. The ground truth for each instance is obtained through the delta debugging-style history partition, and thus it is guaranteed to be 1-minimal. The dataset is available at: github.com/Chenguang-Zhu/DoSC.

An additional evaluation strategy is the development of case studies. We developed case studies on several evolution management tasks in earlier work [20], [21], [23]. We plan to conduct new case studies on additional development tasks, especially with the proposed evolution management framework mentioned in Sect. V.

Ultimately, we intend to evaluate our approach with a comprehensive user study on both experienced and inexperienced developers. The study will help us evaluate the usability of our tool chain and provide further confidence on the effectiveness of our techniques through direct comparisons with manual operations.

VII. Contributions and Status

This PhD work presents a new semantics-based view of software version histories and proposes the use of the semantic history slicing techniques to support various evolution management tasks. We described the work done so far on problem formalization and several alternatives for computing the solutions. Specifically, two semantic slicing algorithms have been proposed and implemented as prototype tools. We demonstrated applications of the proposed techniques in many practical usage scenarios (A1, A2 and A3). The construction of an evolution management framework is still in progress (A4). We also discussed next steps and presented a plan for evaluation.

Acknowledgments

This work is part of a PhD thesis supervised by Prof. Marsha Chechik. Thanks to my supervisor and my collaborators, Julia Rubin and Chenguang Zhu, for their valuable input.

References

[1] Git Version Control System. [Online]. Available: https://git-scm.com

[2] Apache Subversion (SVN) version control system. [Online]. Available: http://subversion.apache.org

[3] J. Rubin, A. Kirshin, G. Botterweck, and M. Chechik, "Managing Forked Product Variants," in *Proc. of SPLC'12*, 2012, pp. 156–160.

[4] K. Muşlu, L. Swart, Y. Brun, and M. D. Ernst, "Development History Granularity Transformations," in *Proc. of ASE'15*, November 2015, pp. 697–702.

[5] R. S. Arnold, *Software Change Impact Analysis*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1996.

[6] D. C. Kung, J. Gao, P. Hsia, F. Wen, Y. Toyoshima, and C. Chen, "Change Impact Identification in Object Oriented Software Maintenance," in *Proc. of ICSM'94*. IEEE Computer Society, 1994, pp. 202–211.

[7] J. Law and G. Rothermel, "Whole Program Path-Based Dynamic Impact Analysis," in *Proc. of ICSE'03*. IEEE, May 2003, pp. 308–318.

[8] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley, "Chianti: A Tool for Change Impact Analysis of Java Programs," in *Proc. OOPSLA'04*. ACM, 2004, pp. 432–448.

[9] L. Zhang, M. Kim, and S. Khurshid, "Localizing Failure-inducing Program Edits Based on Spectrum Information," in *Proc. of ICSM'11*. IEEE, 2011, pp. 23–32.

[10] E. Murphy-Hill, C. Parnin, and A. P. Black, "How We Refactor, and How We Know It," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 5–18, Jan 2012.

[11] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin, "Early Detection of Collaboration Conflicts and Risks," *IEEE Transactions on Software Engineering*, vol. 39, no. 10, pp. 1358–1375, Oct. 2013.

[12] F. Servant and J. A. Jones, "History Slicing," in *Proc. of ASE'11*, 2011, pp. 452–455.

[13] A. Zeller, "Yesterday, My Program Worked. Today, It Does Not. Why?" in *Proc. of ESEC/FSE-7*. Springer-Verlag, 1999, pp. 253–267.

[14] R. Saha and M. Gligoric, "Selective Bisection Debugging," in *Proc. of FASE'17*. Springer-Verlag New York, Inc., 2017, pp. 60–77.

[15] T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller, "Mining Version Histories to Guide Software Changes," in *Proc. of ICSE'04*. IEEE Computer Society, 2004, pp. 563–572.

[16] K. Herzig and A. Zeller, "The Impact of Tangled Code Changes," in *Proc. of MSR'13*. Piscataway, NJ, USA: IEEE Press, 2013, pp. 121–130.

[17] S. Hayashi, T. Omori, T. Zenmyo, K. Maruyama, and M. Saeki, "Refactoring Edit History of Source Code," in *Proc. of ICSM'12*. IEEE, September 2012, pp. 617–620.

[18] G. Rothermel and M. J. Harrold, "Analyzing Regression Test Selection Techniques," *IEEE Transactions on Software Engineering*, vol. 22, no. 8, pp. 529–551, Aug 1996.

[19] J. Ferzund, S. N. Ahsan, and F. Wotawa, "Empirical Evaluation of Hunk Metrics As Bug Predictors," in *Proc. of IWSM'09*, 2009, pp. 242–254.

[20] Y. Li, J. Rubin, and M. Chechik, "Semantic Slicing of Software Version Histories," in *Proc. of ASE'15*, November 2015, pp. 686–696.

[21] Y. Li, C. Zhu, J. Rubin, and M. Chechik, "Semantic Slicing of Software Version Histories," *IEEE Trans. on Software Engineering*, 2017.

[22] Y. Li, C. Zhu, J. Rubin, and M. Chechik, "Precise Semantic History Slicing through Dynamic Delta Refinement," in *Proc. of ASE'16*, September 2016, pp. 495–506.

[23] Y. Li, C. Zhu, J. Rubin, and M. Chechik, "FHistorian: Locating Features in Version Histories," in *Proc. of SPLC'17*, September 2017.

[24] C. Zhu, Y. Li, J. Rubin, and M. Chechik, "A Dataset for Dynamic Discovery of Semantic Changes in Version Controlled Software Histories," in *Proc. of MSR'17*, May 2017, pp. 523–526.