Managing Software Evolution Through Semantic History Slicing

by

Yi Li

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Department of Computer Science
University of Toronto

Abstract

Managing Software Evolution Through Semantic History Slicing

Yi Li
Doctor of Philosophy
Department of Computer Science
University of Toronto
2018

Software change histories are results of incremental updates made by developers. As a byproduct of the software development process, change history is surprisingly useful for understanding, maintaining and reusing software. However, traditional commit-based sequential organization of version histories lacks semantic structure and thus is insufficient for many development tasks that require high-level, semantic understanding of program functionality.

In this dissertation, we propose a new semantics-based view of software version histories, where a set of related changes satisfying a common high level property (also known as *slicing criteria*) is recognized as a *semantic history slice*. As one concrete instantiation, test cases exercising a software functionality can be used as slicing criteria to identify changes implementing the particular functionality.

Specifically, we present a family of automated techniques which analyze the semantics of historical changes and assist developers in many practical software evolution tasks. First, we introduce CSlicer, an efficient static history slicing algorithm which requires only a one-time effort for compilation and test execution. It mostly relies on static analysis of dependencies between change sets and is therefore inexpensive in terms of running time but can be imprecise. Second, we present Definer, a dynamic history slicing algorithm based on delta refinement. It executes tests multiple times and 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 can be much longer. Finally, we describe the design and implementation of a Web-based semantic slicing framework CSlicerCloud which aims to find balance between performance and precision, unifies different history slicing algorithms and provides software developers a flexible and ready-to-use environment for various evolution management tasks.

We have successfully applied the history slicing techniques in many development tasks including creating focused pull requests, back-porting bug fixes, locating feature implementations and building feature models to assist evolution understanding. For validation, we evaluate our approaches on a benchmark of developer-annotated version history instances obtained from real-world open-source software projects from GitHub.

To my parents.

Acknowledgements

Firstly, I would like to express my sincere gratitude to my advisor Marsha Chechik for the continuous support of my PhD studies, for her patience, motivation, and immense knowledge. Her guidance helped me during all of the research and writing of this thesis. I can not imagine having a better advisor and mentor for my PhD studies. Looking back in time, there were many important moments when she showed me good taste for research, taught me how to become a respected researcher, and transformed me into a better person. The very first lesson I learned from her at our first one-on-one meeting is to take effective notes during meetings, which I followed and benefited tremendously from since then. She was always able to identify my weaknesses, relentlessly expose them, and finally give me remedies. During the early days of my PhD, I struggled with presentations and public speaking in general. She persistently listened to the many versions of my talks, gave critiques, and taught me tricks for improving them. It is amazing to see how far I have come with all her generous help.

I have also been privileged to work closely with Julia Rubin. Many of the initial ideas of this thesis came from a discussion with her five years ago. Although we have spent more time talking over Skype than in person, she still played a vital role in shaping and honing the work presented in this thesis. Thank you, Julia.

I am deeply grateful to people who gave me feedback and commented on my thesis. Specifically, I thank the members of my supervising committee, Azadeh Farzan and Andreas Veneris, for their patience and for directing me towards completing the thesis over the years. I owe thanks to Sheila McIlraith and Eric Hehner for carefully reading my thesis and providing insightful comments. I would also like to show my gratitude to my external examiner, Darko Marinov. Darko agreed to serve on my exam committee on short notice and gave me very detailed feedback which really helped clarify and solidify many concepts presented in this thesis.

I was inspired to start doing research and eventually go to graduate school by Jin Song Dong, with whom I did my final year undergraduate research project and a four-month research assistantship before starting my PhD. Jin Song provided me much helpful advice and supported me in many ways during my entire graduate studies. I was also very fortunate to work with many great people in his research group, including Jun Sun, Yang Liu, Tian Huat Tan, Shaojie Zhang, Chunqing Chen, Manchun Zheng, and Manman Chen.

I was very fortunate to have the chance to work with many great mentors who guided me through the journey. Arie Gurfinkel patiently endured my baby steps and taught me many basics when I just started my Master's. Being demanding, criticizing and understanding, he has driven me to success. Aws Albarghouthi has been a role model for me. He answered my naïve questions, showed me command-line tricks, helped me debug, revised my paper drafts, and more importantly, gave me great career advice. In the summer of 2014, I interned with Akash Lal and Shuvendu Lahiri at Microsoft Research in Bangalore, India. I thank them for introducing me to many cool tools and techniques as well as spending time and energy to discuss many immature ideas of mine. In the summer of 2016, I interned with Jayanthkumar Kannan and Gogul Balakrishnan at Google in Mountain View. Jayanth and Gogul were always there for me and they showed me how to engineer rigorous and efficient systems which have high impact in practice.

My excellent colleagues in the Software Engineering group made my graduate school experience enjoyable, and I wholeheartedly thank them for that. Alicia Grubb was the "executive" of the lab and she taught me many things, ranging from interview skills to baking apple pies. Michalis Famelis was such an energetic and inspiring individual. There was no shortage of laughter and debates when Michalis was around. Zachary Kincaid was simply brilliant. It was a joy to listen to Aws and Zak discussing research problems. Chenguang Zhu joined the group in 2015 and made significant contributions to my thesis work. I also appreciate the companionship and friendship of the many people I was fortunate enough to meet and work side-by-side with during my years at University of Toronto: Rick Salay, Alessio Di Sandro, Sahar Kokaly, Federico Mora, Nick Fung, Ramy Shahin, Mike Maksimov, Aamod Kore, Victor Nicolet, Anthony Vandikas, Matthew Amy, Jonathan Eidelman, Lev Naiman, Mahsa Sadi, Benson Quach, Ahmed Mashiyat, Pooya Saadatpanah, Jan Gorzny, Youssef Zaky, Vivien Suen, Marcel Serikawa, Ioanna Stavropoulou, Cynthia Disenfeld, Fabiano Dalpiaz, Juan de Lara, Esther Guerra, and Yanzhen Zou. I really enjoyed our group lunches and board game nights.

I was also lucky to make many great friends who made my graduate studies experience truly wonderful: Xu Huang, Bangbiao Wu, Yongbo Tang, Wenbo Dai, Shuze Zhao, Pantazis Deligiannis, Xinyan Yan, Yan Wang, Sihan Li, Qian Sun, Owolabi Legunsen, Xiaohong Chen, and many others.

To the China Scholarship Council, Greg Wolfond, and Monica Ryckman: thank you! The scholarships I received from them made the last years of my PhD much more comfortable.

Finally, I would like to express my deepest gratitude to my parents, Ruimin and Xiaoe for their unparalleled love. Everything that I have achieved is because of them. I am very thankful to them for enduring my absence over the past eleven years, when other families could reunite during holidays and I was overseas pursuing my studies. Their encouragement and support helped me go through many tough periods and it feels great to share my accomplishments with them.

Yi Li
Toronto, Canada
March 2018

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

"Everything changes and nothing stands still."

— Heraclitus

1.1 Preamble

Real software is seldom created "all at once" and changes are inevitable [Som04]. Software development is typically an incremental and iterative process where many program versions are produced, each advancing and improving on the previous ones. For example, new requirements, errors and bugs emerge while the use cases and the operating environment evolve during the lifespan of a software system. Software developers often take advantage of the knowledge and insights they gain over time to repair, enhance and optimize earlier versions of the system through incremental updates.

Managing a large software system is hard, and managing a fast-evolving one is extremely challenging. The amount of change made to an actively maintained code base can be enormous. According to Potvin and Levenberg [PL16], in 2014, approximately 15 million lines of code (almost equivalent to the size of the Linux kernel) were changed in the Google repository on a weekly basis. As of January 2015, the Google code base had approximately 35 million change sets spanning the company's entire 18-year existence. Our knowledge about the system at any given point of time is no longer relevant, because it becomes obsolete too rapidly.

The history of changes accumulated during the development process has become a crucial artifact for understanding and analyzing software systems from an evolutionary point of view. The information embedded in change sets goes beyond lines added and removed. Modern version control systems record meta data such as timestamps, log messages, developer conversations, and test results along with code edits. Many studies have been conducted to analyze and extract useful information from software change histories. For example, trends and patterns observed in the past changes can be used to estimate quality of new changes [KWZ08] and gain insights about the overall system [WC10].

Existing history-based information retrieval approaches are mostly reduced to *change classification* – given a set of changes whose categories are already known, find a good predictor that identifies the right category for a given new change. The classification approaches use the statistical power of the historical data to separate the buggy changes from the clean ones [WH05, KWZ08], the non-essential changes from the essential ones [KR11], and the unrelated changes from the related ones [HZ13], etc. A similar idea is also used to learn high-level properties of the base system to which changes are applied. Some notable ones include characterizing architectural evolution [ZDZ03, WC10], optimizing development process [BHEN13], and identifying project specific API usage patterns [NHC$^+$16] as well as developer expertise [SJ12, CFSM16].

Dealing with the sheer amount of change data and distilling useful information is only one aspect of effective evolution management. In fact, a wide range of software engineering tasks cannot be easily supported with only the data-mining-style interpretation of historic changes. Particularly, one needs to cope with the constant instability caused by local changes to ensure global reliability. Conversely, one has to break down global functional goals into local steps of individual changes to move forward. These tasks require a clear recognition of the connections between changes and the overall system. The code changes should therefore be put into the context of programing languages and functional requirements with a deeper understanding of the semantics of changes, more precise modeling of system properties, and a rigorous reasoning about their interactions.

Numerous tools and techniques have been proposed to alleviate the gap between local changes and global system properties [Arn96, FRC81, Zel99, PDHP08, YQTR15, LVH10, SFS12]. This dissertation follows this long and rich path and focuses on analysis of program changes in histories. In the rest of this chapter, we look at some related research over the past few decades (Section 1.2). We then state our main contributions and discuss how this dissertation advances the state of software change analysis by contributing novel algorithms and providing support for everyday software developing tasks (Section 1.3).

1.2 Taming Changes in Fast-Evolving Systems

A clear understanding of the interactions between incremental changes and their underlying base system is the prerequisite for many software engineering tasks. For example, developers need to be aware of the consequences and ripple effects of their changes before committing and deploying the changes to the system. On the other hand, when an error is manifested in the system, it is useful to pinpoint the changes that might have caused the failure before a repair is attempted. Over the years, people have developed many techniques such as change impact analysis [Arn96], regression testing [FRC81], upgrade checking [SFS12], and delta debugging [Zel99], to support such activities. In this section, we provide a detailed (but of course incomplete) overview of the recent advances.

As shown in Figure 1.1, a history of changes can be seen as a stack of versions (or snapshots) of the base system, i.e., $\langle v_0, \ldots, v_{i-1}, v_i, v_{i+1}, \ldots, v_n \rangle$, where v_0 is the base version. The set of differences between two consecutive snapshots is called a *change set* (or a *delta*). Essentially, it is a function transforming one snapshot into another. Therefore, a history can also be seen as a sequence of deltas, i.e., $\langle \Delta_1, \ldots, \Delta_i, \Delta_{i+1}, \ldots, \Delta_n \rangle$ and $v_{i+1} = \Delta_{i+1}(v_i)$. There are two streams of research focusing on slightly different aspects of change histories, and we categorize them as *change-centric* and *version-centric* techniques.

1.2.1 Change-Centric Techniques

Change-centric techniques target at a spectrum of program versions and treat deltas as first-class citizens. The goal is to understand relationships between specific changes and their connections to high-level

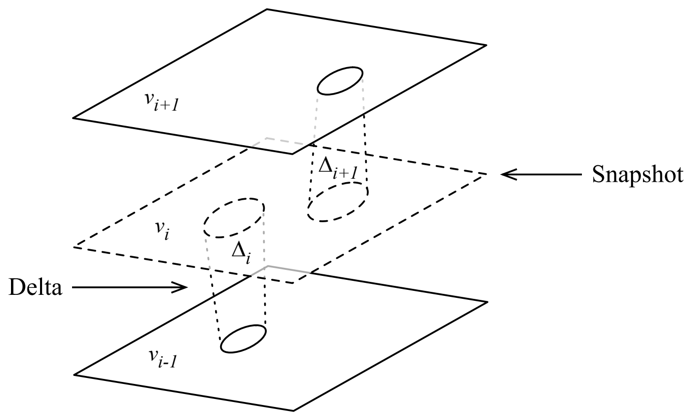


Figure 1.1: Snapshots and deltas.

system properties. Examples include identifying dependencies between changes and deciding whether a change could affect the result of a test suite or cause a bug in the system.

Change Impact Analysis

Change Impact Analysis (IA) [Arn96] solves the problem of determining potential consequences of source code modifications. The consequences are usually given as an impact set containing code elements that might be affected by the change. Existing research on IA can be roughly divided into three categories – the *static* [SLL$^+$15], *dynamic* [LR03, OAL$^+$04] and *combined* [OAH03, RST$^+$04, ZKK11] approaches.

The static IA approaches proposed to date rely on static analysis of program dependencies and call graphs in a style that is very similar to forward program slicing [Wei81]. Like that of any other static technique, the analysis result of static IA is sensitive to various parameters such as the granularity level of changes, and the precision of dependency and pointer analysis [AR11]. Fine tuning parameters to achieve the best cost-effectiveness can be a tricky task. For example, the precision of the control and data dependency used in the slicing algorithm is closely related to the size of the output impact set: an imprecise dependency analysis can be over-conservative, i.e., resulting in large impact sets with many false positives. On the other hand, a more precise algorithm might be too expensive. Acharya and Robinson [AR11] propose a way to combine analyses with different level of precisions to achieve good cost-effectiveness.

The dynamic approaches rely on observable data, for instance, execution traces collected from running a test suite. Dynamic IA techniques are generally more efficient than their static counterparts. But naturally, the impact estimates computed in this manner reflect only program behaviors observed in limited runtime traces; thus, they under-approximate by definition. However, these estimates may be particularly useful in cases where safety is not required because they provide more precise information about impact – relative to a specific program behavior – than static techniques [OAL$^+$04].

Some recent work uses the combined approach. For example, Ren et al. [RST$^+$04] 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 input. First, it builds dynamic call graphs for both versions, before and after the changes, using the test suite. Then it computes a set of atomic changes defined at the method level by comparing the two program snapshots. Finally, it uses the old call graph along with the classified atomic changes to predict the tests that might be affected, and uses the new call graph to select the affecting changes that might cause test failures. FaultTracer [ZKK11] improved the precision of Chianti by extending the standard dynamic call graph with field access information.

Regression Testing

As a way of reestablishing confidence in software correctness, *regression testing* [FRC81] reruns previously completed tests whenever a change is made to a piece of software. We say that a change introduces a regression fault if any one of the regression test cases fails.

Regression testing can be expensive if the test suite is large and tests take a long time to run. *Regression test selection and prioritization* techniques [GHK$^+$01, KP02, YH12, GMS$^+$14] analyze incremental changes to a code base and choose to run only those tests whose behaviors may be affected by the latest changes in the code. By focusing on a small subset of the regression test suite, the testing process runs faster and can be more tightly integrated into the development process. Substantial research has been done on this topic [ERS10], and many of the approaches use ideas similar to change impact analysis.

Gligoric et al. [GMS$^+$14, GEM15] considered a more practical model for change histories and generalized traditional IA-like test selection techniques to distributed version histories. Software change histories in a distributed Software Configuration Management (SCM) system often have many branches; therefore, they can be modeled as directed acyclic graphs. Their novel treatment of branch merging commands allowed selecting test sets which are safe and substantially smaller than the ones computed by previous techniques.

The power of traditional regression testing in exposing regression faults is limited by the quality of the existing test suites. Due to the high cost of building thorough test suites, the tests used in practice tend to exercise only a small fraction of the program's functionality and may be inadequate for testing changes in a program. Behavioral regression testing [OX08, JOX10] identifies behavioral differences between two versions of a program through dynamic analysis. First, it generates a large number of test inputs that focus on the changed parts of the code. Second, it runs the generated test inputs on the old and new versions of the code and identifies differences in the observable behaviors. Finally, it analyzes the identified differences and presents them to developers. With additional tests generated, behavioral regression testing can potentially expose more bugs due to faulty changes.

Delta Debugging

Software debugging and, specifically, fault localization, is one of the most time-consuming processes in software development [AA14]. Programmers spend significant amount of time locating root causes of failures. *Delta debugging* [Zel99, ZH02] is one of the automated fault localization techniques that simplify and isolate a minimal cause of a given test failure by analyzing historic changes.

Given two program versions and a test suite such that the tests pass in the older version and fail in the newer one, delta debugging finds an approximated minimal set of changes (deltas) that contribute to the test failure. The high-level idea is to repeatedly partition the changes and opportunistically reduce the search space when the target tests fail on one of the partitions. The divide-and-conquer-style of search continues with increasing partition granularities until a *1-minimal* [ZH02] partition is reached. The 1-minimal set produced guarantees to manifest the erroneous behaviors whereas removing any single change from it would not. The algorithm applies to program inputs as well; it can also be used to find small failure-inducing inputs for a buggy program.

Later, Misherghi and Su [MS06] proposed *Hierarchical Delta Debugging* (HDD) – an extension of the original delta debugging idea which considers the structures of the changed data and improves performance as well as solution quality. Instead of treating input space as a flat atomic list, HDD assumes tree-structured data and performs partitions with the underlying structures in mind. The HDD algorithm has an arguably higher chance of success in finding the right partitions for structured inputs such as programs, HTML/XML files, video codecs, and UI interactions.

1.2.2 Version-Centric Techniques

Version-centric techniques mostly concern the properties of particular snapshots of a system in the presence of changes. The umbrella term *differential program analysis* [WE03] is used for analysis of two related programs in order to determine their functional difference or prove their semantic equivalence. The programs considered can either be two versions in a temporal order or even two distinct representations of the same program, e.g., source code vs. binary. Using one program as a specification, one could establish relative correctness for the other.

Differential Symbolic Execution

Person et al. introduced the concept of *Differential Symbolic Execution* (DSE) [PDHP08] which is an adaptation of the classic symbolic execution technique [Kin76] for analyzing program changes. DSE is capable of characterizing the semantics of a set of program changes in terms of *behavioral difference*. In contrast to a source code *line difference* which consists of a set of added and removed lines and carries little program semantics, a behavioral difference is a more precise description of the variations in how program executes due to changes. Behavioral difference can be used to produce better version differencing information. For example, formatting and minor syntactic changes may not alter the meaning of a program but cannot be distinguished by line differences. Subsequent maintenance activities (such as program documentation, regression testing, fault localization and program summarization) that rely on change analysis to identify the effects of a change may, consequently, do large amounts of unnecessary work. DSE performs standard symbolic execution on both program versions, before and after the change, and either reports that the two versions are equivalent or characterizes the behavioral differences by identifying the sets of inputs that cause different effects.

Directed incremental Symbolic Execution (DiSE) [PYRK11] builds on DSE by adding static impact analysis for locating possible locations where the execution may vary. The insight of DiSE is to leverage the information extracted from the cheaper change impact analysis to enable more efficient symbolic execution of programs as they evolve. Godefroid et al. introduced *incremental compositional symbolic execution* [GLRG11], and the key novelty is compositionality – the search process can be made compositional by memorizing symbolic execution sub-paths as test summaries which are reusable during the search, resulting in a search algorithm that can be exponentially faster than a non-compositional one [God07].

Later, Cadar and Palikareva proposed *shadow symbolic execution* [CP14] which runs two different program versions in the same symbolic execution instance, with the old version effectively “shadowing” the new one, in order to effectively drive execution toward the behavior that has changed from one version to the other. Instead of relying on static analysis for locating changed locations, shadow symbolic execution uses precise runtime information about the execution of both versions for identifying divergent behaviors. This could provide important opportunities for pruning large parts of the program space and potentially significantly improve the scalability of differential symbolic execution techniques.

Relative Specification and Verification

As an alternative to the behavior-based change semantics, *relative specification* is another mechanism for capturing differences between two program versions. By specifying properties over variables from both programs, one can explicitly express intended equivalences or differences as precise logical formulas and therefore obtain verifiable relative correctness.

In 2012, Yi et al. introduced *software change contracts* [QYR12, YQTR15]. They designed a customized JML-like [BCC$^+$05] change contract language in order to write specifications for changes in Java programs. Unlike conventional program contract languages which specify pre- and post-conditions of methods, the change contract language is able to describe how post-conditions of the same method in two consecutive versions relate to each other, under certain pre-conditions. Exceptional behavior, as well as structural changes (e.g., insertion and removal of parameters or fields) and refactoring, are also supported.

Differential Assertion Checking (DAC) [LMSH13], proposed by Lahiri et al., uses *mutual specifications* to specify relative safety. The goal is to check whether the updated version is as safe as the old version whose safety had been previously established. Given two program snapshots v_0 and v_1, and a set of assertions all of which hold at v_0, DAC checks whether these assertions still hold at v_1. Unlike in DSE, behavioral preservation does not have to be guaranteed across versions, and only a weaker form of equivalence with respect to assertions is checked.

When verifying the correctness of relative specifications, both change contracts and differential assertion checking use a similar approach of constructing a *composed program* [LMSH13, YQTR15]. A composed program incorporates both versions of the target program along with their relative specifications, so that verification conditions can be generated and verified by a theorem prover using standard software verification techniques [DKW08]. The verified mutual specifications can then serve as a certificate for the relative safety of the updated program.

Incremental Upgrade Checking

Given two program versions and a safety proof for the original one, *incremental upgrade checking* [SFS12] adapts the earlier proof artifacts to prove or disprove safety for the updated version. In essence, it solves a similar problem as DAC, attempting to prove a property-specific equivalence of two program versions: whether both versions satisfy a property P. However, incremental upgrade checking focuses on reducing verification costs, while DAC and other relative specification verification techniques emphasize establishing cross-version relationships.

Sery et al. [SFS12] proposed an algorithm based on *function summaries*. Their algorithm maintains a function summary (a logical formula which constrains the inputs and outputs of a function) for each procedure such that the summary over-approximates the behaviors of the procedure. When a newer version arrives, one can locally check if the summaries are still valid over-approximations of the updated procedure. The local checks are much cheaper compared with re-verification of the whole program. If the check succeeds, then the upgrade is safe. If not, the check has to be propagated upwards to the caller of the modified functions. As soon as safety is reestablished, new summaries are generated for all functions for future reference. If the check fails at the root of the call tree, then the upgrade is unsafe, and an error trace is reported.

Later, Fedyukovich et al. [FGS14] introduced a new algorithm which reuses *safe inductive invariants* from earlier proofs. An *inductive invariant* ψ for the original version v_0 is a labeling of its Control Flow Graph (CFG), so that ψ (1) contains all initial states, and (2) is closed under the transition relation for program v_0. The inductive invariant ψ is also *safe* if it does not intersect with an error state. The proposed upgrade verification algorithm attempts to construct a new safe inductive invariant φ for the updated version v_1 by first weakening ψ to π such that π is inductive for v_1, and then strengthening π to φ such that φ is also safe for v_1. If such a φ can be found, then it effectively becomes a safety proof for v_1. The algorithm is applied to verify the correctness of compiler optimizations.

Yang et al.'s *property differencing* work [YKPR14] also aims to reduce verification costs for incremental checking, but approaches the problem from a different angle. Rather than reusing proof artifacts, they partition the analysis of code and property specifications and compute logical differences between the initial and updated properties. Then they rely on change impact analysis to reduce the number of properties needed to be re-verified. Property differencing is complementary to other reuse-based techniques and can be applied in combination with them to achieve better overall performance.

1.2.3 Gaps Identified in This Thesis

While several solutions, from both the change-centric and version-centric world, have been proposed to establish the connections and interactions between changes and the underlying base systems, a number of important software evolution management tasks remain challenges for practitioners.

The History Restructuring Challenge

Software Configuration Management (SCM) systems, such as Git [Git16b], SVN [SVN16] and Mercurial [Mer16], are commonly used for hosting software development artifacts. They allow developers to periodically submit their ongoing work, storing it as an increment over a previous version. Such an increment is usually referred to as a *commit* (Git and SVN) or a *change set* (Mercurial). 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 at different customers.

However, the sequential and manually-managed organization of changes is inflexible and lacks support for many tasks that require high-level, semantic understanding of program functionality [RKBC12]. Moreover, manually-managed change histories often mix together changes made for multiple development activities, such as feature implementations, bug fixes, and performance improvements [HZ13, MSBE15], which is suboptimal for many development scenarios. Thus, when submitting pull requests for review, contributors should refrain from including unrelated changes as suggested by many project contribution guidelines [Bit16, Lin17, Azu16]. Also, project maintainers commonly suggest to split a tangled code patch into multiple ones, e.g., as indicated by these code review comments:[1]

[1] https://github.com/apache/commons-lang/pull/41

"Okay, nevertheless we need to split this up because it is unrelated to the issue we're talking about."

Therefore, it is necessary to be able to restructure change histories according to the specific development scenarios at hand. In particular, being able to group and collapse, categorize and split changes with precise semantic understanding can enable better project organization, easier code maintenance, and more effective history analysis.

The Change Isolation and Migration Challenge

Developers often need to locate and migrate functionality, either for porting bug fixes between branches or for propagating features from development branches to release branches. Back-porting is one example of such a migration, when changes made in a newer version of software are ported to an earlier one in order to provide the updated functionality to all users. Several SCM systems provide the 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.

Even in very disciplined projects, when such commits can be identified by browsing their associated log messages, the functionality of interest might depend on another functionality implemented on the same branch. To ensure correct execution, that other functionality has to be identified and migrated to the new branch as well which is a tedious and error-prone manual task [RKBC12]. For example, consider the feature "make Groovy method blacklist truly append-only", introduced in version 1.3.8 of the Elasticsearch project [Ela15] – a real-time distributed data search and analytics framework written in Java. This feature and its corresponding test cases were implemented in a single commit (#`647327f4`). Yet, propagating this commit to a different branch will likely fail because one of the added statements makes use of a field whose declaration was introduced in an earlier commit (#`64d8e2ae`).

Thus, it would be ideal to have a technique which identifies all dependencies of the target functionality so that when they are isolated and migrated together, the resulting product preserves the desired functionalities and behavioral correctness.

The Variability Reverse Engineering Challenge

Successful software evolves, more and more often, from a single system to a set of system variants tailored to meet the similar yet different functional requirements from distinct markets and clients. *Software Product Line Engineering* (SPLE) [PBL05, RC13b] is a software development paradigm which promotes systematic software reuse by leveraging the knowledge about the set of available features (also known as *core assets* [CN01]), relationships among the features (also known as *feature models* [Bat05]) and relationships between the features and software artifacts that implement them. This systematic organization of the product portfolio represents variabilities explicitly and reduces the chance of code duplications; therefore, it helps decrease development and maintenance effort and cost.

However, since it is difficult to foresee the needs of reuse a priori (and hence to design a software product line upfront) in reality, software product variants often emerge ad-hoc, by means of duplication. This is also known as the "clone-and-own" approach – a routinely used practice where developers create a new product by copying/branching an existing variant and later modifying it independently from the original [DRB+13]. The reverse engineering process from an existing loosely organized product portfolio to systematic SPLE approaches relies on the identification of the relationship between the features and their corresponding implementation, which is the main goal of *feature location* techniques [DRGP13, RC13b].

Recently, a number of techniques for identifying features in the Software Product Line (SPL) context have been proposed [XXJ12, ZFdSZ12, ASH+13b, ASH+13a, LAG+14, LLE13, TKW+13, LLHE16]. Most of such techniques are based on intersecting code of multiple product variants in order to identify code fragments shared by variants with a particular feature. The identified code fragments can then be attributed to that feature. These *intersection-based techniques* [LLHE16] operate in a static manner and are effective when a large number of product variants (tens or even hundreds) are available.

Yet, we often cannot rely on the availability of a large number of variants, especially in the "clone-and-own" scenario. Such cloned products are commonly maintained in a version control system, e.g., Git [Git16b]. Their number can be relatively small, e.g., 3-10 products. For such scenarios, the problem of variability reverse engineering from software change histories remains a challenge.

1.3 Contributions of This Thesis

A common root of the challenges presented in the previous section is the missing connections between individual changes and the high-level system semantics. In this dissertation, we consider the problem of managing fast-evolving software systems with a new semantic-based view of version histories, where a set of related changes satisfying a common high level functional property (*slicing criteria*) is known as a *semantics-preserving history slice*. We hypothesize that the proposed organization of version histories is effective in software evolution management tasks including software understanding, maintenance, and reuse.

Figure 1.2 outlines our major contributions and high-level relationship between them. Major contributions are shown in bold and surrounded by boxes along with their descriptions listed below. Some contributions also come with automated tool support which we built to evaluate our techniques in practical evolution management scenarios. Arrows between boxes indicate dependencies between different parts. The following discussion explicates each individual contribution in detail.

Semantic History Slicing. Specifically, we studied the day-to-day evolution management practices of open source software projects and identified many use cases of our proposed semantic-based history organization. We formally defined the problem of *semantic history slicing* for software version histories (see Chapter 3). Given a base program v_0, its change history H, and a test suite T, semantic history slicing computes a portion of H – a *semantic slice* H', such that after replaying H' on top of v_0, the functionalities captured in T are still preserved. Additionally, a *minimal semantic slice* is often more desired in practical applications. We then developed two automated semantic history slicing techniques including a *static* slicing approach and a *dynamic* slicing approach.

Static Slicing Technique. The static approach relies on the identification of the *functional* and *compilation* dependencies with respect to the target slicing criteria. Functional dependencies correspond to the changes that can directly affect the core functionality of interest. They are essential for ensuring behavioral correctness of the migrated code. On the other hand, compilation dependencies are the changes required by language rules to guarantee well-formedness of the functional core. The generic static history slicing algorithm is independent of the underlying SCM infrastructure and tools. We also extended the

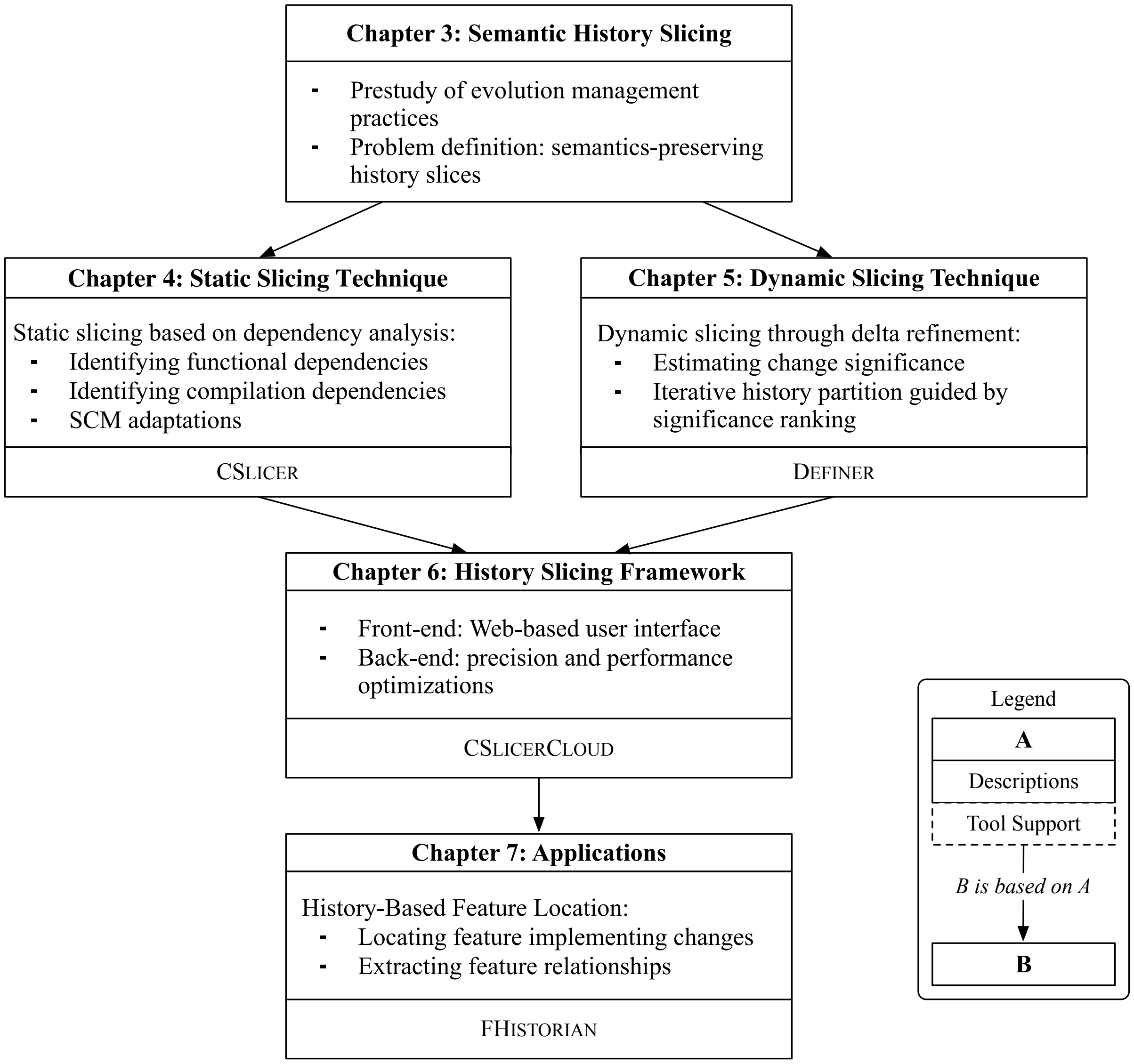

Chapter 3: Semantic History Slicing
Prestudy of evolution management practices
Problem definition: semantics-preserving history slices
Chapter 4: Static Slicing Technique
Static slicing based on dependency analysis:
Identifying functional dependencies
Identifying compilation dependencies
SCM adaptations
CSlicer
Chapter 5: Dynamic Slicing Technique
Dynamic slicing through delta refinement:
Estimating change significance
Iterative history partition guided by significance ranking
Definer
Chapter 6: History Slicing Framework
Front-end: Web-based user interface
Back-end: precision and performance optimizations
CSlicerCloud
Chapter 7: Applications
History-Based Feature Location:
Locating feature implementing changes
Extracting feature relationships
FHistorian
Legend
A
Descriptions
Tool Support
B is based on A
B

Figure 1.2: The contributions of this dissertation and dependencies between them.

generic algorithm to bridge the gap between language semantic entities and text-based modifications, thus making it applicable to existing text-based SCM systems. We implemented a prototype tool CSlicer and applied it to several case studies on real world software projects.

Dynamic Slicing Technique. The dynamic approach operates based on the concept of *change significance* which measures the importance of a specific change with respect to the target slicing criteria. We proposed the *dynamic delta refinement* algorithm which estimates significance of changes and then uses the significance ranking of changes to guide the iterative history partitions for reaching a minimal semantic history slice. The dynamic delta refinement algorithm produces shorter history slices than the static history slicing algorithm. Our prototype implementation Definer was evaluated on a benchmark of version history instances collected from GitHub.

History Slicing 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 built a Web-based history slicing service framework CSlicerCloud. 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.

Applications in Evolution Management. We successfully applied the semantic history slicing techniques in many evolution management tasks including creating more focused pull requests (Chapter 4), back-porting bug fixes (Chapter 4), locating feature implementations and building feature models to assist evolution understanding (Chapter 7).

- *Creating Focused Pull Requests.* The first use case of semantic slicing is creating logically clean and easy-to-merge pull requests. This directly addresses the "history restructuring challenge". 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.

- *Porting Functionalities Across Versions.* Another important 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. This directly addresses the "change isolation and migration challenge". 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.

- *Identifying Features and Feature Relationships.* Finally, to address the "variability reverse engineering challenge", we applied semantic history slicing for identifying features and feature relationships in cloned product line variants. This 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. The results show that semantic slicing can be an effective way of locating feature-implementing changes in software version histories with the presence of feature tests. In addition, the generated 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.

1.4 Organization

The rest of this dissertation is organized as follows:

- In Chapter 2, we formalize the main concepts required for describing our contributions.
- In Chapter 3, we look at the challenge of identifying semantically-related changes in histories from an evolution management angle and provide a formal definition for the problem of *semantic history slicing*.
- In Chapter 4, we present a static approach, CSlicer, for computing semantic history slices based on change dependency analysis. (Based on [LRC15, LZRC17b].)
- In Chapter 5, we present a dynamic approach, Definer, for computing semantic history slices based on delta refinement. (Based on [LZRC16].)
- In Chapter 6, we describe the design and implementation of CSlicerCloud, a Web-based framework which unifies our semantic history slicing algorithms and implements many important optimizations. (Based on [LZRC18].)
- In Chapter 7, we present a case study of applying semantic history slicing for improving one of the most important evolution management tasks – feature location. (Based on [LZRC17a].)
- Finally, in Chapter 8, we summarize our contributions and discuss open problems and future research directions.

Appendix A contains details about a dataset of version controlled histories for dynamic discovery of semantic changes, which is used in various parts of this dissertation. (Based on [ZLRC17].)

Chapter 2

Background

In this chapter, we present a number of core concepts required for describing our contributions in the rest of this dissertation.

2.1 Programs

To keep the presentation of algorithms concise, we step back from the complexities of the full Java language and concentrate on its core object-oriented features. We adopt a simple functional subset of Java from *Featherweight Java* [IPW01], denoting it by P.

2.1.1 Language Syntax

The syntax rules of the language P are given in Figure 2.1. Many advanced Java features, e.g., interfaces, abstract classes and reflection are stripped from P, while the typing rules which are crucial for the compilation correctness are retained [KA08].

$$\begin{array}{rcl}
P & ::= & \overline{L} \\
L & ::= & \texttt{class } C \texttt{ extends } C\{\overline{C\ f}\texttt{; } K\ \overline{M}\} \\
K & ::= & C(\overline{C\ f})\{\texttt{super}(\overline{f})\texttt{; } \overline{\texttt{this.}f = f\texttt{;}}\} \\
M & ::= & C\ m(\overline{C\ x})\{\texttt{return } e\texttt{;}\} \\
e & ::= & x \mid e.f \mid e.m(\overline{e}) \mid \texttt{new } C(\overline{e}) \mid (C)e
\end{array}$$

Figure 2.1: Syntax rules of the P language [IPW01].

We say that p is a *syntactically valid program* of language P, denoted by $p \in P$, if p follows the syntax rules. A program $p \in P$ consists of a list of class declarations ($\overline{L}$), where the overhead bar $\overline{L}$ stands for a (possibly empty) sequence $L_1, \ldots, L_n$. We use $\langle\rangle$ to denote an empty sequence and comma for sequence concatenation. We use $|L|$ to denote the length of the sequence. Every class declaration has *members* including *fields* ($\overline{C\ f}$), *methods* ($\overline{M}$) and a single *constructor* (K). A method *body* consists of a single *return* statement; the returned expression can be a variable, a field access, a method lookup, an instance creation or a type cast.

$$C <: C \qquad \frac{C <: D \quad D <: E}{C <: E} \qquad \frac{\texttt{class } C \texttt{ extends } D\texttt{\{...\}}}{C <: D}$$

Figure 2.2: Subtyping rules of the P language [IPW01].

The subtyping rules of P, shown in Figure 2.2, are straightforward. We write $C <: D$ when class C is a subtype of D. As in full Java, subtyping is the reflexive and transitive closure of the immediate subclass relation implied by the `extends` keyword. The field and method lookup rules are slightly different from the standard ones (see Figure 2.3) – field *overshadowing* and method *overloading* are not allowed while method *overriding* is allowed in Featherweight Java [IPW01]. For example, when resolving a method call $C.m$, the method list $\overline{M}$ of class C is first consulted. If m is defined in C then its type and body are returned as a pair $(\overline{B} \to B, \overline{x}.e)$. Otherwise, the lookup continues recursively on the super class of C. Note that $\textsc{Methods}(m, C)$ is a partial function. Since in P, the special class `Object` is assumed to have no method, $\textsc{Methods}(m, \texttt{Object})$ is undefined.

$$\textsc{Fields}(\texttt{Object}) = \langle\rangle$$

$$\frac{\texttt{class } C \texttt{ extends } D\texttt{\{}\overline{C\ f}\texttt{;}\ K\ \overline{M}\texttt{\}} \quad \textsc{Fields}(D) = \overline{D\ g}}{\textsc{Fields}(C) = \overline{D\ g}, \overline{C\ f}}$$

$$\frac{\texttt{class } C \texttt{ extends } D\texttt{\{}\overline{C\ f}\texttt{;}\ K\ \overline{M}\texttt{\}} \quad B\ m(\overline{B\ x})\texttt{\{return } e\texttt{;\}} \in \overline{M}}{\textsc{Methods}(m, C) = (\overline{B} \to B, \overline{x}.e)}$$

$$\frac{\texttt{class } C \texttt{ extends } D\texttt{\{}\overline{C\ f}\texttt{;}\ K\ \overline{M}\texttt{\}} \quad m \notin \overline{M}}{\textsc{Methods}(m, C) = \textsc{Methods}(m, D)}$$

Figure 2.3: Fields and methods lookup rules of the P language [IPW01].

2.1.2 Abstract Syntax Trees

A valid program $p \in P$ can be parsed as an *abstract syntax tree* (AST), denoted by $\textsc{Ast}(p)$. We adopt a simplified AST model where the smallest entity nodes are fields and methods. Formally, $r = \textsc{Ast}(p)$ is a rooted tree with a set of nodes $V(r)$. The root of r is denoted by $\textsc{Root}(r)$ which represents the compilation unit, i.e., the program p. Each entity node x has an identifier and a value, denoted by $id(x)$ and $\nu(x)$, respectively. In a valid AST, the identifier for each node is unique (e.g., fully qualified names in Java) and the values are canonical textual representations of the corresponding entities. We denote the parent of a node x by $\textsc{Parent}(x)$.

For example, Figure 2.4 shows an AST for the program `Foo.java`. The following facts are true about the node `f`,

$$id(\texttt{f}) = \text{“}\texttt{foo.B.f(int)}\text{”},$$
$$\nu(\texttt{f}) = \text{“}\texttt{static int f(int x)\{return x-1;\}}\text{”},$$
$$\textsc{Parent}(\texttt{f}) = \texttt{B}.$$

```java
class A {
  int x;
  int h()
  {return B.f(x);}
  // hunk deps
  int g()
  {return (new B()).y;}
}
class B {
  int y = 0;
  static int f(int x)
  {return x - 1;}
}
```

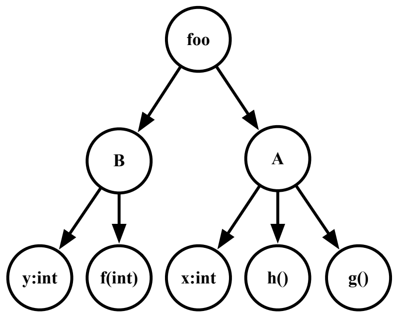


Figure 2.4: `Foo.java` and its AST representation.

The children are unordered – the ordering of child nodes is insignificant. Therefore, each program has its unique AST representation.

2.2 Program Semantics

This dissertation mainly concerns about the behavioral semantics of programs, which can be effectively captured by test executions.

2.2.1 Tests and Test Suites

We assume that semantic functionalities can be captured by tests and the execution trace of a test is deterministic [RH96]. For simplicity, a test can be abstracted into two parts – the setup code which initializes the testing environment and executes the target functionalities using specific inputs, as well as the oracle checks which verify that the produced results match with the expected ones. A test execution succeeds if all checks pass.

Definition 1. *(Test). 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.*

Definition 2. *(Test Suite). A test suite is a collection of tests that can exercise and demonstrate the functionality of interest.*

Let test suite T be a set of test cases $\{t_i\}$. We write $p \models T$ if and only if program p passes all tests in T, i.e., $\forall t \in T \cdot t(p)$.

2.2.2 Dynamic Invariants

Dynamic invariants [PE04] are likely invariants that are discovered from program executions. They assert predicates that hold during the execution at specific program points including procedure entries and exits, and aggregate program points of multiple class instances. We are particularly interested in three types of predicates:

- method preconditions asserting values of input parameters,

- method postconditions asserting returned values, and
- all values taken by fields throughout the execution.

A wide range of dynamic invariants is detected and reported by Daikon [EPG+07], but only a subset of the invariants are used in this dissertation. In particular, we consider a subset of the invariants which involve a single program variable, including comparisons with constants (e.g., `x == K`, `x == K1 (mod K2)`, `K1 <= x <= K2`, `x != null`), single-valuedness (e.g., `x has only one value`), and value range (e.g., `x one of {a,b}`).

Given two invariant sets I and I', the *invariant delta*, $I \setminus I'$, consists of all invariants in I that are not implied by any invariant in I'. Formally, $I \setminus I' = \{i \in I | \neg(\exists i' \in I' \cdot i' \Rightarrow i)\}$.

2.3 Changes and Change Histories

Based on the program representations used, changes and change histories can be defined as differences between either structured ASTs or unstructured plain texts.

2.3.1 AST-Based View

Let Γ be the set of all ASTs. Now we define changes, change sets and change histories as AST transformation operations.

Definition 3. *(Atomic Change). An atomic change operation $\delta : \Gamma \nrightarrow \Gamma$ is a partial function which transforms $r \in \Gamma$ producing a new AST r' such that $r' = \delta(r)$. It can be either an* insert*, * delete *or* update *(see Figure 2.5).*

An insertion $\textsc{Ins}((x, n, v), y)$ inserts a node x with identifier n and value v as a child of node y. A deletion $\textsc{Del}(x)$ removes node x from the AST. An update $\textsc{Upd}(x, v)$ replaces the value of node x with v. A change operation is *applicable* on an AST if its preconditions are met. For example, the insertion $\textsc{Ins}((x, n, v), y)$ is applicable on r if and only if $y \in V(r)$. Insertion of an existing node is treated the same as an update.

$$\frac{y \in V(r)}{\begin{array}{cc} V(r') \leftarrow V(r) \cup \{x\} & \textsc{Parent}(x) \leftarrow y \\ id(x) \leftarrow n & \nu(x) \leftarrow v \end{array}} \ \textsc{Ins}((x, n, v), y)$$

$$\frac{x \in V(r)}{V(r') \leftarrow V(r) \setminus \{x\}} \ \textsc{Del}(x) \qquad \frac{x \in V(r)}{\nu(x) \leftarrow v} \ \textsc{Upd}(x, v)$$

Figure 2.5: Types of atomic changes [FG06]. Refer to Section 2.1.2 for AST-related symbols.

Definition 4. *(Change Set). Let r and r' be two ASTs. A change set $\Delta : \Gamma \nrightarrow \Gamma$ is a sequence of atomic changes $\langle \delta_1, \ldots, \delta_n \rangle$ such that $\Delta(r) = (\delta_n \circ \cdots \circ \delta_1)(r) = r'$, where $\circ$ is standard function composition.*

A change set $\Delta = \Delta_{-1} \circ \delta_1$ is applicable to r if δ_1 is applicable to r and Δ_{-1} is applicable to $\delta_1(r)$. Change sets between two ASTs can be computed by tree differencing algorithms [CRGW96]. For instance, in Figure 2.6, the change set C_3 consists of an insertion of a new node `y` to `B` followed by an update of the node `g`.

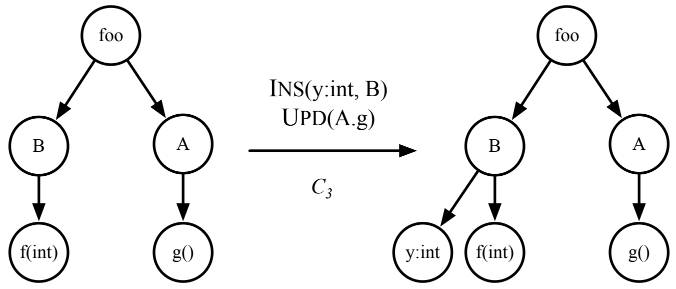


Figure 2.6: Visualizing C_3 as a sequence of atomic changes applied on ASTs.

Definition 5. *(Change History). A history of changes is a sequence of change sets, i.e.,* $H = \langle \Delta_1, \dots, \Delta_k \rangle$.

Definition 6. *(Sub-history). A sub-history is a sub-sequence of a history, i.e., a sequence derived by removing change sets from* H *without altering the ordering.*

We write $H' \lhd H$ indicating H' is a sub-history of H and refer to $\langle \Delta_i, \dots, \Delta_j \rangle$ as $H_{i..j}$. The applicability of a history is defined similar to that of change sets. We use $SH(H)$ to denote the set of all sub-histories of H.

2.3.2 Text-Based View

Another view of changes used by text-based SCM tools is called *hunk* [FAW09, LRC15]. A hunk is a group of adjacent or nearby line insertions or deletions with surrounding context lines which stay unchanged. Let P be the set of all program texts.

Definition 7. *(Hunk). A hunk* $\hat{\delta} : P \nrightarrow P$ *is a partial function which transforms* $p \in P$ *producing a new program text* p' *such that* $p' = \hat{\delta}(p)$.

For example, Figure 2.7 shows a hunk of one line deletion and two line insertions, marked by "-" and "+", respectively. The *context* (the lines not marked by "-" or "+" in Figure 2.7) that comes with a hunk is useful for ensuring that the hunk can be applied at the correct location even when the line numbers change for the target program texts.

```
      // hunk deps
      int g()
-     {return 0;}
+     {return (new B()).y;}
    }
    class B {
+     int y = 0;
      static int f(int x)
      {return x - 1;}
```

Figure 2.7: Hunk representation of C_3.

A *conflict* happens if the context cannot be matched when applying a hunk. In the current example, the maximum length of the contexts is four lines: up to two lines before and after each change.

A *commit* is a collection of hunks, in no particular order, which takes a program text p and transforms it to produce a new program text $\hat{\Delta}(p)$. Applying a commit is equivalent to composing its corresponding hunks, each representing a set or line changes with an approximate locality. More formally, a commit $\hat{\Delta}$ is defined as follows.

Definition 8. *(Commit). Let p and p' be two program texts. A commit $\hat{\Delta} : P \nrightarrow P$ is a set of hunks $\{\hat{\delta}_0, \ldots, \hat{\delta}_n\}$ such that $\hat{\Delta}(p) = (\hat{\delta}_0 \circ \cdots \circ \hat{\delta}_n)(p) = p'$, where $\circ$ is standard function composition.*

Definition 9. *(Commit History). A commit history is a sequence of commits, i.e., $H = \langle \hat{\Delta}_1, \ldots, \hat{\Delta}_k \rangle$.*

For simplicity, in the rest of this thesis, we use the same notations for both AST-based and text-based changes where the context is clear.

Chapter 3

Semantic History Slicing – the Evolution Management Challenge

3.1 Introduction

In Chapter 1, we argued that the sequential and manually-managed organization of changes is inflexible and lacks support for many tasks that require high-level, semantic understanding of program functionality such as locating feature implementations and porting hot fixes. To address these challenges, we propose to use well-organized tests as identifiers for corresponding software functionalities.

In this chapter, we first present a prestudy of the evolution management practices observed in open-source software project development, which emphasizes the necessity of bringing semantics to analyzing version histories (Section 3.2). Then we give a formal definition of *semantics-preserving slice*, and discuss a few general approaches for finding such history slices (Section 3.3). We conclude this chapter in Section 3.4.

3.2 Prestudy

In this prestudy, we investigate the common practices followed in a series of software evolution management tasks including preparing candidates for new releases and submitting pull requests for review. To understand how developers approach those tasks, the recommended best practices as well as the gaps between the two, we analyzed qualitatively data from multiple sources including developer conversations, project contribution guidelines and change histories of several open-source software projects. The main findings are summarized as follows.

3.2.1 The Pitfalls of Cherry-Picking

We start with a conversation among a team of Bazel [Baz17] developers which took place on their GitHub issue page.[1] Bazel is an open-source multi-language build system backed by Google. The project team has a convention of releasing a new version of the build tool every month and opening a new issue for the release on GitHub. This particular release, v0.4.3, happened in December 2015. The conversation was

[1] Accurate as of September 22, 2017: `https://github.com/bazelbuild/bazel/issues/2246`

among three team members, "damienmg", "iirina" and "petemounce", discussing the preparation of a release candidate.

damienmg: Filing the release bug for this month since I haven't seen it yet.

iirina: Created RC: ... and pushed to GitHub: ... waiting for ci.bazel.io to finish.

damienmg: Release failed. Might be because of my last changes investigating.

damienmg: Filed #2256.

petemounce: Please ping me when rc is up and I'll publish to chocolatey.

damienmg: The RC is up but the mail wasn't send because of a bug in the debian publishing :(. You can do the chocolatey package from the 0.4.3rc2 URL at ...

petemounce: ok - published and available at ...

iirina: Damien commited a fix for #2256 which I have cherry picked when creating and pushing RC2 on Friday evening. However the release still failed and @katre identified the commit as breaking the normal build process for non-releases, so the change got reverted.

damienmg: The reason for the breakage is because the manifest is empty. Easy enough to fix and rollforward. I think the release is safe without it because it's only breaking the CI pipeline on non release trigger.
However this did not fixed the underlying issue: the section is empty in the DSC file. I need more investigation to fix it.

*damienmg: **To be cherry-picked: 4a75349.***

*damienmg: **4a75349 55c97bc are to be cherry-picked**. Still one change that is being submitted plus the fix for the distribution artifact.*

*damienmg: Ok so far the remaining bug seems like just tests setup fixes. **I think we can go forward with another RC with the following fixes: 4a75349 4a75349 55c97bc 39e5a46 4fb378c**.*

iirina: I tried to create the new RC: ... but it failed: ...

*iirina: **There was a conflict in src/test/shell/bazel/bazel_coverage_test.sh**. I merged the file and ... However the script exited before and the release didn't finish. The cherry-picks added after the one with the merge conflict were not added to the release.*

iirina: Did "$ cherry-pick –continue; exit", after already cherry-picking and it accidentally deleted the release branch: ... Re-did the whole RC creation and this time it was successfully and it includes all cherry picks, only now it is marked again as RC1 :(...

iirina: Fixed candidate name: ...

iirina: Pushing now "$ scripts/release/release.sh push", waiting for ci.bazel.io to finish.

*iirina: Email was sent to bazel-discuss. **The release notes in the email are wrong, they just include the commit message of the last cherry pick.** The web release notes seem nicer, although the cherry picks are not nicely formatted and **it still seems 2/6 cherry picks are missing**.*

*damienmg: **missing cherry-pick: acbcbc2 (forgot it)**.*

*iirina: **Creating new candidate with additional cherry-pick acbcbc2** ... Created 0.4.3rc4 on branch release-0.4.3. Pushed: ...*

iirina: New email was sent to bazel-discuss. The release notes look really good in the email, including almost all (7/9) cherry-picks (except 4a75349 and 39e5a46). The web release notes are still not nicely formatted.

iirina: ***Should we also cherry-pick 4975760?*** *It fixes #2247 (someone accidentally added a usage of Java8-only code and some users can't built bazel on a Linux system with JDK7).* ***The culprit is ca99bb7 which is included in the candidate, but the fix isn't.*** *However we should actually release today since I'll be OOO starting tomorrow.*

damienmg: We cannot release before Friday or at least before tomorrow. The JDK breakage is on the test right? In which case is not really important.

damienmg: ***Oh it is not on the test so we should cherrypick it.*** *Strange it was able to bootstrap...*

iirina: OK then I will cherry pick it now and will release Friday evening from home.

damienmg: Don't do the release from home. We'll find someone to take over.

iirina: Created and pushed RC5 including cherry-pick 4975760.

iirina: Seems like bazel-install failed with Jenkins errors: ... will re-run.

iirina: The email was sent to bazel-discuss. All is well except ***there are still 2/10 cherry picks missing (the same as before).***

iirina: I re-run the build and got the same Jenkins errors: ... platform is linux-x86_64

damienmg: ***Let's finally cherry-pick 05fd076 today and do the release tomorrow evening so we still have one working day before Christmas ...***

The central issue in this conversation is that "iirina" failed to create a new release candidate by manually cherry-picking commits from the development branch to the release branch. Later, even with the help from another team member, "damienmg", who has much better expertise in the project, they still experienced many difficulties such as accidentally causing merge conflicts, losing log messages, and missing essential commits due to negligence. After many attempts (highlighted in bold and lasting about 10 days), the new version was finally released before Christmas.

This developer conversation is a good demonstration of the common frustrations faced by many developers who rely on the cherry-picking functionality of version control tools for evolution management tasks. The pitfalls of cherry-picking are rooted in the lack of semantic understanding of changes. For example, it is very difficult for developers who are not familiar with the project and its change histories (such as "iirina") to identify all the changes relevant to a maintenance task. The lack of semantic understanding is also reflected in how difficult it is to discover relationships between commits. For instance, without knowing that the commit `#4975760` is a fix to the culprit commit `#ca99bb7` which causes a JDK breakage, "iirina" missed the fix commit at the beginning and the cherry-picked release candidate ended up being buggy.

Interestingly, to get a working release candidate, the team took a trial-and-error approach. For them, there is a well-defined standard for a successful release candidate. That is, all intended features should be included, there should not be any merge conflict, the tests have to pass, and the release notes should be formatted correctly. Their release creation process is as follows:

Step 1 Cherry-pick commits to a new release branch.

Step 2 Run the release publishing script which tests the release candidate.

Step 3 If the release candidate does not meet the requirements, go back to Step 1 and try a different set of commits.

Notice that this resembles an algorithm for automatically finding a set of commits for a successful release, except that there is human involved at Step 1. An omniscient expert may pick the correct commits and the process will finish in one go. A naïve person may make mistakes all the time, but eventually he/she will find the correct ones after enough attempts. If we are to design an algorithm mimicking this process and eliminate a human from the loop, there is always a trade-off between spending effort in making smart choices in Step 1, and being lazy in Step 1 and waiting for more iterations. We will come back to the discussion of this trade-off many times in the rest of this dissertation.

3.2.2 Tests as Semantic Indicators

We saw in the Bazel team case that the passing of tests is one of the criteria of success for a release candidate. Tests are also commonly used as *semantic indicators* in many other evolution management tasks. For example, in regression testing [FRC81], tests are used to check if a new change to the software breaks already established system properties. In dynamic feature location [WS95], tests are used to identify the place where a feature is implemented in the code base. These techniques rely on the ability of tests in exercising and validating semantics of the software.

Tests are also widely available in well-organized software projects. It is often a strict requirements for all new functionalities implemented to be accompanied by test cases. For example, the "Guide to Developing Maven [Mvn16]" requires all patches relating to functionalities to be associated with tests:

> *"It is expected that any patches relating to functionality will be accompanied by unit tests and/or integration tests."*

and the Git contribution guidelines [Git16c] also emphasize the adequacy of tests in terms of the ability of triggering new behaviors:

> *"Make sure that you have tests for the bug you are fixing. When adding a new feature, make sure that you have new tests to show the feature triggers the new behavior when it should, and to show the feature does not trigger when it shouldn't."*

Therefore, using tests as semantic indicators in evolution management is very practical in well-organized large projects, where the needs for better semantic understanding of version histories are also the greatest.

Generally speaking, for the purpose of exercising and validating program behaviors, we do not impose restrictions on the types of tests, as long as they meet the requirements in Section 2.2.1 – being deterministic and producing Pass/Fail results. For example, unit tests, integration tests, and system tests can all be used as semantic indicators. However, unit tests which exercise more fine-grained program behaviors are more suitable for identifying functionality-related changes in practice. Ultimately, the quality of a test as a semantic indicator is decided by how complete and focused it is in exercising the intended behaviors. This coincides with the guidelines for writing good unit test cases [Git16c].

3.2.3 Patches Should Always Be Focused

Let us look at another conversation between two JUnit [JUn16] developers, "PeterWippermann" and "kcooney", regarding a pull request from the former.[2]

[2] https://github.com/junit-team/junit4/pull/1348

peterwippermann: As discussed in #1338 I refactored some of runners to be able to reuse their functionality. This mainly included the conversion of private instance methods to public (static) class methods. When functionality was hidden in other methods I also introduced new methods (e.g. Parameterized.normalizeParameter(Object)).

kcooney: ***Could you please remove all the unnecessary reformatting? It makes it hard to review...***

peterwippermann: @kcooney How should I remove the formatting? Sorry, but Eclipse always formats the whole file and it's not my fault that this file wasn't formatted accordingly before :-/

kcooney: Tell Eclipse to not format files on save (or, if possible to only reformat changes lines). I realize the files might not follow the current formater settings, ***but reviewing code that is reformatted in the same pull is painful****.*

After working on refactoring of some code, "PeterWippermann" submitted the changes as a pull request in order for the changes to be reviewed and merged into the shared repository. However, the code reviewer suggested that all changes unrelated to the topic of the pull request be dropped, because they make the reviewing "painful". In fact, it is a common practice to keep the changes as logically concentrated as possible, as specified in many project contribution guidelines.

For example, such requirements can be found in "Microsoft Azure: Ways to Contribute [Azu16]":

> *"In order to speed up the process of accepting your contributions, you should try to make your checkins as small as possible, avoid any unnecessary deltas and the need to rebase."*

in "How to Contribute to BitCoin Core [Bit16]":

> *"Patchsets should always be focused. For example, a pull request could add a feature, fix a bug, or refactor code; but not a mixture. Please also avoid super pull requests which attempt to do too much, are overly large, or overly complex as this makes review difficult."*

and in "Submitting Patches: the Essential Guide to Getting Your Code into the Kernel [Lin17]":

> *"Separate each logical change into a separate patch. For example, if your changes include both bug fixes and performance enhancements for a single driver, separate those changes into two or more patches. If your changes include an API update, and a new driver which uses that new API, separate those into two patches."*

These guidelines strongly urge that patches (or pull requests) should be focused: a patch should make changes in exactly one logical unit. For instance, a patch adding a new feature should never be mixed with a bug fix, and a patch on feature A should never touch feature B, etc. The reasons for doing so are to keep things small and simple. Small patches are less prone to conflicts and thus easier to merge. Simple patches are easier to review and can speed up the progress of the development.

To create small and simple patches, we need ways to separate unrelated changes according to certain standards. In the JUnit case, the culprits were reformatting changes and they are easy to track with the help of code editors. However, in many other cases, there is no simple solution for us to logically separate changes according to their semantics. In Section 3.3, we propose a new semantics-based view of version histories as an attempt to address this issue.

3.2.4 Study Summary

To summarize, developers constantly need to pick and move functionalities, and most of the time they do so by moving commits in version histories. For example, this happens when they create release candidates and submit pull requests for review. Yet, SCM systems nowadays fail to support these maintenance tasks with semantic understanding of changes, which makes the process very time consuming and error prone.

On the other hand, tests are used and strictly required in many software projects to exercise and validate new functionalities. This makes them very good semantic indicators for identifying changes relevant to a functionality – a set of changes are said to preserve a functionality if they pass the corresponding tests.

Finally, in many practical evolution management settings, one needs to locate changes which preserve a target functionality and at the same time are as focused as possible. Therefore, to support these tasks of finding semantically relevant changes, we need to design techniques which are precise and include as little irrelevant changes as possible.

3.3 Problem Definitions

Inspired by the concept of *program slicing* [Tip95], we refer to a sub-history which preserves certain semantic properties of the original history as a *semantics-preserving slice*. The preserved semantic properties are referred to as *slicing criteria*. In this dissertation, we focus on using tests as one concrete instantiation of the slicing criteria.

3.3.1 Semantics-Preserving Slices

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

Definition 10. *(Semantics-preserving Slice). A semantics-preserving slice of history H with respect to T is a sub-history $H' \lhd H$ such that the following properties hold:*

1. $H'(p_0) \in P$,
2. $H'(p_0)$ *is well-typed,*
3. $H'(p_0) \models T$.

The problem of semantic history slicing is to identify such a *semantics-preserving slice* given change history H and target tests T. We sometimes refer to semantics-preserving slice as *semantic slice* for short. A trivial and 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,

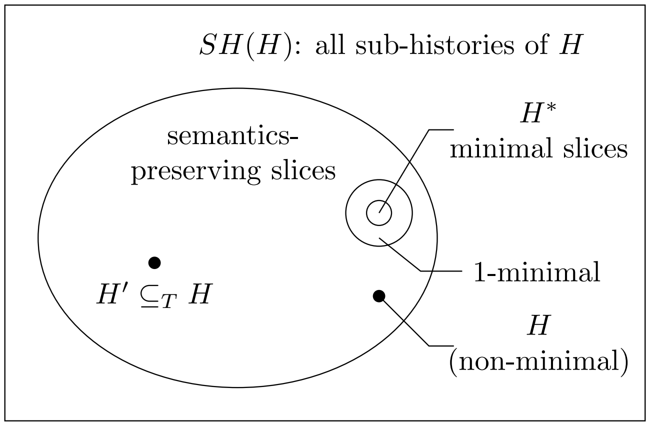


Figure 3.1: Relationships between various history slices.

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 [LZRC17b] here.

An optimal algorithm which runs the test only once cannot exist in any case: in order to determine whether to keep a change set or not, it needs to at least be able to answer the decision problem, "given a fixed program p and test t, for any arbitrary program p', will the outputs of t be different on both?" which is known to be undecidable [RH94].

Therefore, we are often more interested in an approximation to the optimal solution, which can be efficiently computed in practice. We say a sub-history H^* of H is a *minimal semantic slice* if H^* is semantics-preserving and it cannot be further shortened without losing the semantics-preserving properties (see Definition 10).

Definition 11. *(Minimal semantics-preserving slice). Let H^* be semantics-preserving, i.e., $H^* \lhd_T H$. H^* is a minimal semantic slice of H if, $\forall H_{sub} \subset H^* \cdot H_{sub} \not\models T$.*

Definition 12. *(1-Minimal Semantic Slice). Let H^* be semantics-preserving, i.e., $H^* \lhd_T H$. H^* is a* 1-minimal semantic slice *of H if, $\forall \delta \in H^* \cdot (H^* \setminus \{\delta\}) \not\models T$.*

As shown in Figure 3.1, 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, 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.

3.3.2 Finding Semantics-Preserving Slices

Since optimal solutions are often intractable for the problem of finding semantics-preserving slices, this dissertation explores two types of semantic history slicing techniques which trade-off optimality for efficiency, namely the *static* and *dynamic* history slicing approaches. We briefly introduce them here and leave the details for Chapters 4 and 5, respectively.

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 (see Chapter 4) 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, producing non-minimal slices.

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 (see Chapter 5) 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* [Zel99]. 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 are guaranteed to be *1-minimal* [ZH02] – removing any single commit from the history slice will break the desired semantic properties.

3.4 Summary

In this chapter, we studied the common practices followed in managing software evolution with existing SCM systems. We identified a number of challenges faced by developers such as creating release candidates through manual cherry-picking and submitting logically focused pull requests. We then proposed semantic history slicing in order to address those challenges by bringing version histories and program semantics together. We formally defined the concept of semantics-preserving history slices and briefly described two possible approaches of finding such history slices.

Chapter 4

Static History Slicing Based on Dependency Analysis

“You show the world as a complete, unbroken chain, an eternal chain, linked together by cause and effect.”

Hermann Hesse

4.1 Introduction

As discovered in the prestudy presented in Section 3.2, text-based version histories are cumbersome for many evolution management tasks which require good understanding of the high-level program semantics. The problem of identifying the exact minimal subset of a version history that implements a particular functionality of interest is referred to as *semantic history slicing*. Computing the optimal semantics-preserving history slice requires enumerating all sub-histories and is not scalable in practice.

One can make the problem more tractable by designing algorithms which compromise on slice optimality but still produce sound and reasonably small approximations of the optimal slice. In this chapter, we look at a efficient static approach for finding semantics-preserving history slices, called CSlicer. CSlicer has two main phases. The first phase is generic and independent of any specific SCM system in use. It relies on change dependency analysis techniques to conservatively identify all atomic changes in the given history that contribute to the *functional* and *compilation* correctness of the functionality of interest. The second phase adapts the output produced in the first one to specifics of SCM systems. More precisely, it 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, e.g., SVN [SVN16] or Git [Git16b]. This phase can optionally be skipped when using language-aware merging tools [Sem16] or in semantic-based SCM systems [Cow16]. However, such systems are not dominant in practice yet.

The use cases of the CSlicer system include but are not limited to porting functionalities; it can also be used for refactoring existing branches, e.g., by reorganizing branches as functionality-related ones.

Contributions

We summarize this chapter's contributions as follows:

- We propose a generic static semantic slicing algorithm that is independent of underlying SCM infrastructures and tools.
- We extend the generic algorithm to bridge the gap between language semantic entities and text-based modifications, thus making it applicable to existing text-based SCM systems.
- We instantiate the overall approach, CSlicer, by providing a fully-automatic semantic slicing tool applicable for Git projects implemented in Java. The source code of the tool, as well as binaries and examples used in this paper, are available at `https://bitbucket.org/liyistc/gitslice`.
- We evaluate the tool on a number of real-world medium- to large-scale software projects. The results show that our approach allows to identify functionality-relevant subsets of original histories that (a) correctly capture the functionality of interest while being (b) substantially smaller than the original ones.

Organization

This chapter is organized as follows:

- In Section 4.2, we illustrate CSlicer with a simple example.
- In Section 4.3, we formalize the static history slicing approach and prove its correctness.
- In Section 4.4, we describe implementation and optimization details.
- In Section 4.5, we report on our experimental findings.
- Finally, we discuss related work and conclude the chapter in Sections 4.6 and 4.7, respectively.

4.2 CSlicer by Examples

In this section, we illustrate CSlicer on a simple schematic example inspired by the feature migration case in the Elasticsearch project [Ela15] mentioned earlier in Section 1.2.3.

Example 1

Figure 4.1 shows a fragment of the change history between versions v1.0 and v1.1 for the file `Foo.java`. Based on this, we introduce a few concepts used in the rest of this chapter.

Input History

Initially, as shown in version v1.0, the file contains two classes, `A` and `B`, each having a member method `g` and `f`, respectively.

Later, in change set C_1, a line with a textual comment was inserted right before the declaration of method `A.g`. Then, in change set C_2, the body of `B.f` was modified from `{return x+1;}` to `{return`

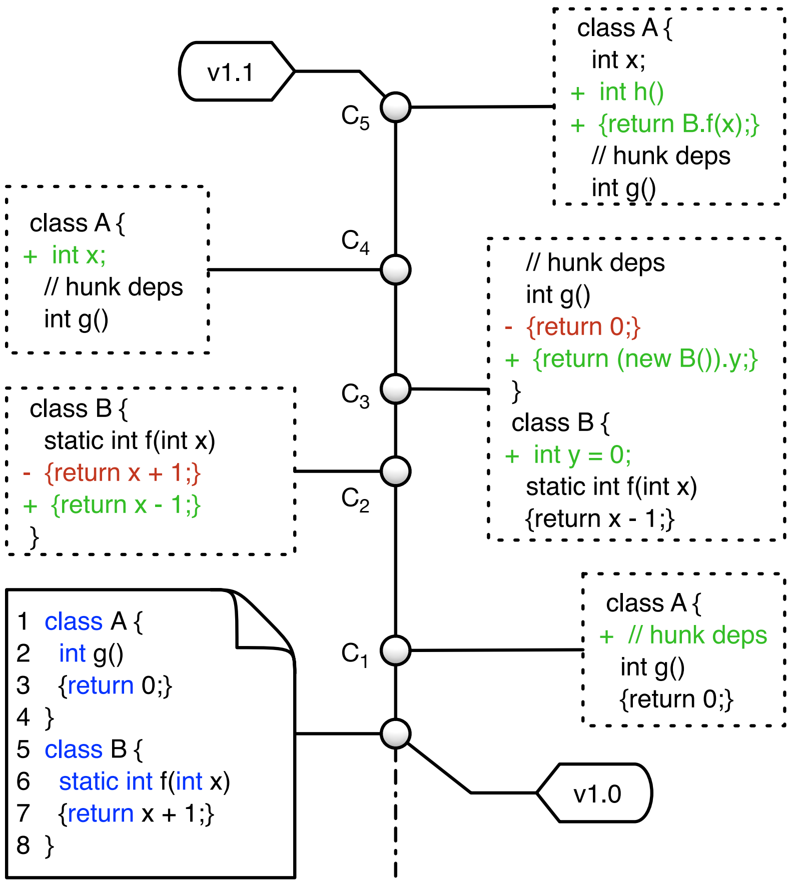


Figure 4.1: Change history of `Foo.java` in Example 1.

`x-1;}`. In change set C_3, the body of `A.g` was updated to return the value of a newly added field `y` in class `B`. In change set C_4, a field declaration was inserted in class `A` and, finally, in change set C_5, a new method `h` was added to class `A`. The resulting program in v1.1 is shown in Figure 4.2a on the left.

Each dashed box in Figure 4.1 encloses a commit written in the *unified format* (the output of command `diff -u`). The lines starting with "`+`" are inserted while those starting with "`-`" are deleted. Each bundle of changed lines is called a *hunk* and comes with a *context* – a certain number of lines of surrounding text that stay unchanged. In Figure 4.1, these are the lines which do not start with "`+`" or "`-`". The context that comes with a hunk is useful for ensuring that the change is applied at the correct location even when the line numbers change. A *conflict* is reported if the context cannot be matched. In Example 1, the maximum length of the contexts is four lines: up to two lines before and after each change.

Slicing Criteria

Suppose the functionality of interest is that the method `A.h`() returns "-1". This functionality was introduced in C_5 and now needs to be back-ported to v1.0. Simply cherry-picking C_5 would result in failure because,

- the body of method `B.f` was changed in C_2 and the change is required to produce the correct result;
- the declaration of field `A.x` was introduced in C_4 but was missing in v1.0, which would cause

```
class A {
  int x;
  int h()
  {return B.f(x);}
  // hunk deps
  int g()
  {return (new B()).y;}
}
class B {
  int y = 0;
  static int f(int x)
  {return x - 1;}
}
```

(a) `Foo.java` at v1.1.

```
class A {
  int x;
  int h()
  {return B.f(x);}
  // hunk deps
  int g()
  {return 0;}
}
class B {
  static int f(int x)
  {return x - 1;}
}
```

(b) `Foo.java` at v1.0 with the sliced sub-history $\langle C_1, C_2, C_4, C_5 \rangle$.

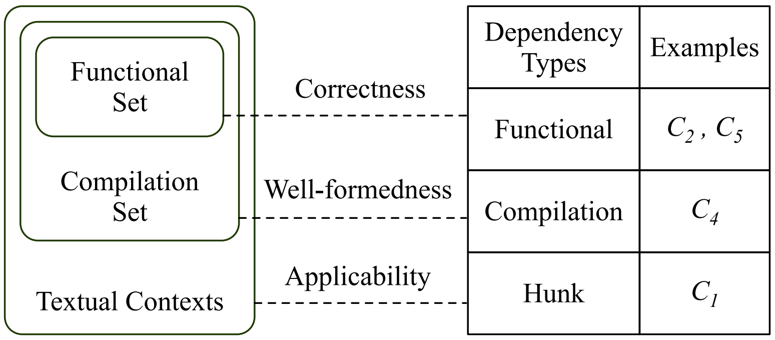
Figure 4.2: `Foo.java` before and after semantic slicing in Example 1.

compilation errors; and

- a merge conflict would arise due to the missing context of C_5 – the text that appears immediately after the change. This text was introduced in C_1.

Computing Semantic Slice

In fact, the change histories form a dependency hierarchy with respect to the target functionality (see Figure 4.3). At its core, the *functional set* contains program components which directly participate in the test execution to deliver the target functionality, e.g., methods `A.h` and `B.f`.



Dependency Types	Examples
Functional	C_2, C_5
Compilation	C_4
Hunk	C_1

Figure 4.3: Change dependency hierarchy.

Step 1: Functional Dependencies. To start with, CSlicer examines the history and identifies *functional dependencies* that are essential for the semantic correctness of the functional set, e.g., C_2, C_5.

Step 2: Compilation Dependencies. In addition, CSlicer computes the *compilation set* which connects the functional core with its structural supporting components, i.e., classes `A`, `B` and the field declaration `int x` in `A`. Similarly, the corresponding contributing changes are called the *compilation dependencies*, e.g., C_4. They are necessary to guarantee program well-formedness including syntactic correctness and type safety.

Step 3: Hunk Dependencies. Finally, to ensure the selected changes can be applied using a text-based SCM system, some additional changes which provide textual contexts should be included as well. We call these changes the *hunk dependencies*, e.g., C_1. In the semantic slicing phase, our proposed algorithms compute a set of commits that are required for porting the functionality of interest to v1.0 successfully: $\{C_1, C_2, C_4, C_5\}$. This process is formalized in Section 4.3.

Slicing Result

Applying the set of commits in sequence on top of v1.0 produces a new program shown in Figure 4.2b on the right. It is easy to verify that the call to `A.h` in both programs returns the same value. Changes introduced in commit C_3 – an addition of the field `B.y` and a modification of the method `A.g` – do not affect the test results and are not part of any other commit context. Thus, this commit can be omitted. The final semantic slicing result is a sub-history of four commits $\langle C_1, C_2, C_4, C_5 \rangle$.

4.3 The CSlicer Approach

In this section, we present the static history slicing algorithms in detail.

4.3.1 Overview of the Workflow

We start with the high level overview of our approach.

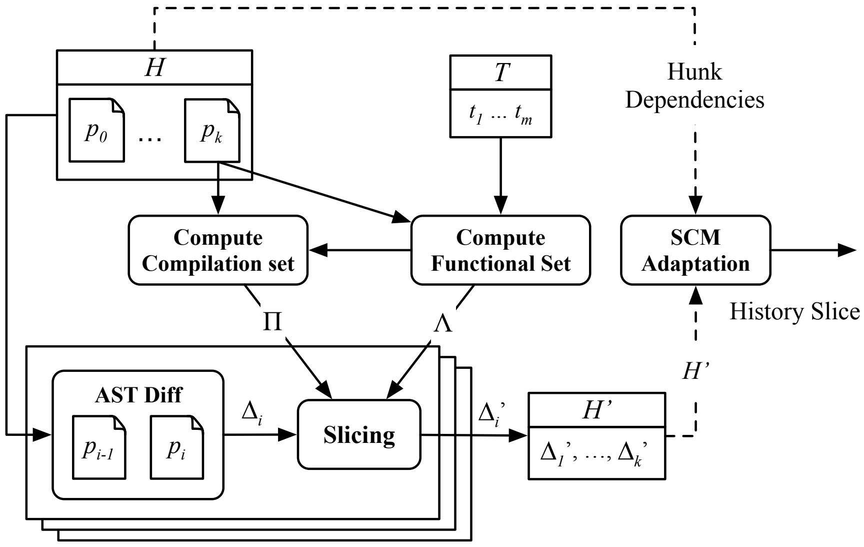


Figure 4.4: High-level overview of the semantic slicing algorithms.

Figure 4.4 illustrates the high-level workflow of the semantic slicing algorithms. First, the functional set (Λ) and compilation set (Π) are computed based on the latest version p_k and the input tests T. The original version history H is then distilled as a sequence of change sets $\langle \Delta_1, \dots, \Delta_k \rangle$ through AST differencing. This step removes cosmetic changes (e.g., formatting, annotations, and comments) and only keeps in Δ_i atomic changes over code entities. Each such set Δ_i then goes through the core slicer component which decides whether to keep a particular atomic change or not. This component outputs a sliced change set Δ_i', which is a subsequence of Δ_i. Finally, the sliced change sets are concatenated and returned as a sub-history H'. Optionally, a post-processing step (SCM Adaptation) of H' is needed if the sliced history is to be applied using text-based SCM systems. Below we describe each step in turn, illustrating them through the running example presented in Section 4.2.

Step 1: Computing Functional Set

CSlicer executes the test on the latest version of the program (left-hand side of Figure 4.2a), which triggers method `A.h`. It dynamically collects the program statements traversed by this execution. These include the method bodies of `A.h` and `B.f`. The set of source code entities (e.g., methods or classes) containing the traversed statements is called the *functional set*, denoted by Λ. The functional set in the current example is $\{\texttt{A.h}, \texttt{B.f}\}$. Intuitively, if (a) the code entities in the functional set and (b) the execution traces in the program after slicing remain unchanged, then the test results will be preserved. Special attention has to be paid to any class hierarchy and method lookup changes that might alter the execution traces, as discussed in more detail in Section 4.4.2.

Step 2: Computing Compilation Set

To avoid causing any compilation errors in the slicing process, we also need to ensure that all code entities referenced by the functional set are defined even if they are not traversed by the tests. Towards this end, CSlicer statically analyzes all the reference relations based on p_k and transitively includes all referenced entities in the *compilation set*, denoted by Π. The compilation set in our case is $\{\texttt{A}, \texttt{A.x}, \texttt{B}\}$. Notice that the classes `A` and `B` are included as well since the fields and methods require their enclosing classes to be present.

Step 3: Change Set Slicing

In the change set slicing stage, CSlicer iterates backwards from the newest change set Δ_k to the oldest one Δ_1, collecting changes that are required to preserve the "behavior" of the functional and compilation set elements. Each change is divided into a set of *atomic changes* (see Definition 3). Having computed the functional and compilation set (highlighted in Figure 4.1), CSlicer then goes through each atomic change and decides whether it should be kept in the sliced history (H') based on the entities changed and their change types. In our example, C_2 and C_5 are kept in H' since all atomic changes introduced by these commits – `B.f` and `A.h` – are in the functional set. C_4 contains an insertion of `A.b` which is in the compilation set. Hence, this change is also kept in H'. C_3 can be ignored since the changed entities are not in either set.

During the slicing process, CSlicer ensures that all entities in the compilation set are present in the sliced program, albeit their definitions may not be the most updated version. Because the entities in the compilation set are not traversed by the tests, differences in their definitions do not affect the test results.

Optional: SCM Adaptation

In the SCM adaptation phase, change sets in H' are mapped back to the original commits. As some commits may contain atomic changes that sliced away by the core slicing algorithm, including these commits in full can introduce unwanted side-effects and result in wrong execution of the sliced program. We eliminate such side-effects by reverting unwanted changes. That is, we automatically create an additional commit that reverts the corresponding code entities back to their original state. In addition, we compute hunk dependencies of all included commits and add them to the final result as well. For example, the comment line added in C_1 forms a context for C_5. Therefore, C_1 is required in the sliced history to avoid merge conflicts when cherry-picking C_5. The details of this process are discussed in Section 4.3.3.

4.3.2 The Semantic Slicing Algorithm

Now we present in detail the semantic slicing algorithm which is independent from the underlying SCM systems and it follows essentially the workflow depicted in Figure 4.4. The optional SCM adaptation phase will be discussed in Section 4.3.3.

Algorithm Descriptions

Algorithm 1 The semantic slicing algorithm.

```
Require: $|H| > 0 \wedge H(p_0) \in P \wedge H(p_0) \models T$
Ensure: $H' \subseteq H \wedge H'(p_0) \in P \wedge H'(p_0) \models T$
1:  procedure SemanticSlice($p_0, H, T$)
2:      $H', k \leftarrow \langle\rangle, |H|$                          ▷ initialization
3:      $p_k \leftarrow H(p_0)$                                  ▷ $p_k$ is the latest version
4:      $\Lambda \leftarrow$ FuncDep($p_k, T$)                          ▷ functional set
5:      $\Pi \leftarrow$ CompDep($p_k, \Lambda$)                         ▷ compilation set
6:      for $i \in [k, 1]$ do                                   ▷ iterate backwards
7:          $\Delta'_i \leftarrow \langle\rangle$                              ▷ initialize sliced change set
8:          for $\delta \in \Delta_i$ do
9:              if $\neg$Lookup($\delta, H_{1..i}(p_0)$) then                  ▷ keep lookup
10:                 if $\delta$ is Del $\vee\ id(\delta) \notin \Pi$ then
11:                     continue                                ▷ skip non-comp and deletes
12:                 if $\delta$ is Upd $\wedge\ id(\delta) \notin \Lambda$ then
13:                     continue                                ▷ skip non-test updates
14:             $\Delta'_i \leftarrow \Delta'_i, \delta$                         ▷ concatenate the rest
15:         end for
16:         $H' \leftarrow H', \Delta'_i$                               ▷ grow $H'$
17:     end for
18:     return $H'$
19: end procedure
```

The main SemanticSlice procedure is shown in Algorithm 1. It takes in the base version p_0, the original history $H = \langle\Delta_1, \ldots, \Delta_k\rangle$ and a set of test cases T as the input. Then it computes the functional and compilation set Λ and Π, respectively (Lines 4 and 5).

FuncDep(p_k, T)**.** Based on the execution traces of running T on p_k, the procedure FuncDep returns the set of code entities (AST nodes) traversed by the test execution. This set (Λ) includes all fields assigned and all methods (and constructors) called during runtime. Although not allowed in Featherweight Java, field declarations with explicit inline initializations are also included when analyzing real Java code, because these also involve assignments to fields.

CompDep(p_k, Λ)**.** The procedure CompDep analyzes *reference* relations in p_k and includes all code entities referenced by the elements of Λ into the compilation set Π. We borrow the set of rules for computing Π from Kästner and Apel [KA08], where the authors formally prove that their rules are

$$\frac{C <: D \quad C \in \Pi}{D \in \Pi}\ [\text{L.1}] \qquad \frac{f : C \in \Pi}{C \in \Pi}\ [\text{L.2}]$$

$$\frac{C(\overline{D\ f})\{\texttt{super}(\overline{f})\texttt{;}\overline{\texttt{this}.f = f\texttt{;}}\} \in \Pi}{C \in \Pi \quad \overline{D} \in \Pi \quad \overline{f} \in \Pi}\ [\text{K1}] \qquad \frac{C\ \ m(\overline{D\ x})\{\texttt{return}\ e\texttt{;}\} \in \Pi}{C \in \Pi \quad \overline{D} \in \Pi}\ [\text{M1}]$$

$$\frac{\ldots\{\texttt{return}\ e.f\texttt{;}\} \in \Pi}{f \in \Pi}\ [\text{E1}] \qquad \frac{\ldots\{\texttt{return}\ e.m(\overline{e})\texttt{;}\} \in \Pi}{m \in \Pi}\ [\text{E2}]$$

$$\frac{\ldots\{\texttt{return new}\ C(\overline{e})\texttt{;}\} \in \Pi}{C \in \Pi}\ [\text{E3}] \qquad \frac{\ldots\{\texttt{return}\ (C)e\texttt{;}\} \in \Pi}{C \in \Pi}\ [\text{E4}]$$

$$\frac{x \in \Pi}{\textsc{Parent}(x) \in \Pi}\ [\text{P1}] \qquad \frac{x \in \Lambda}{x \in \Pi}\ [\text{T1}]$$

Figure 4.5: CompDep reference relation rules.

complete and ensure that no reference without a target is ever present in a program. Applying these rules, which are given in Figure 4.5 and described below, allows us to guarantee type safety of the sliced program.

L1 a class can only extends a class that is present;

L2 a field can only have type of a class that is present;

K1 a constructor can only have parameter types of classes that are present and access to fields that are present;

M1 a method declaration can only have return type and parameter types of classes that are present;

E1 a field access can only access fields that are present;

E2 a method invocation can only invoke methods that are present;

E3 an instance creation can only create objects from classes that are present;

E4 a cast operation can only cast an expression to a class that is present;

P1 an entity is only present when the enclosing entities are present;

T1 an entity is in the compilation set if it is in the functional set.

We iterate backwards through all the change sets in the history (Lines 6-17) and examine each atomic change in the change set. An atomic change δ is included into the sliced history if it is an insertion or an update to the functional set entities, or an insertion of the compilation set entities. Updates to the compilation set entities (that are not in the functional set) are ignored since they generally do not affect the test results.

Our language P does not allow method overloading or field overshadowing, which limits the effects of class hierarchy changes. Exceptions are changes to subtyping relations or casts which might alter method lookup (Line 9). Therefore we define function Lookup to capture such changes,

$$\textsc{Lookup}(\delta, p) \triangleq \exists m, C \cdot \textsc{Methods}(m, C) \neq \textsc{Methods}'(m, C),$$

where $\textsc{Methods}$ and $\textsc{Methods}'$ are the method lookup function for p and $\delta(p)$, respectively. Finally, the sliced history H' is returned at Line 18.

Correctness of Algorithm 1

Assume that every intermediate version of the program p is syntactically valid and well-typed. We show that the sliced program p' produced by the $\textsc{SemanticSlice}$ procedure maintains such properties.

Lemma 1. *(Syntactic Correctness). $H'(p_0) \in P$.*

Proof. From the assumption, every intermediate version $p_0, \ldots, p_k$ is syntactically valid. As a result, their ASTs are well-defined and every change operation $\delta \in H$ is applicable given all preceding changes. Updates on tree nodes do not affect the tree structure and, therefore, do not have any effect on the preconditions of the changes. We can safely ignore updates when considering syntactic correctness.

We prove the lemma by induction on the loop counter i. The base case is when $i = k$ and $H' = \langle\rangle$. By definition, $H'(p_k) = p_k$ is in P. Assume that $H' \circ H_{1..i}(p_0) \in P$. We must show that $(H', \Delta_i') \circ H_{1..i-1}(p_0) \in P$. From the condition on Lines 10 and 12, we know that changes affecting only the entities outside of Π are ignored. So for any change $\delta \in H'$, we have $id(\delta) \in \Pi$. Depending on the change type of δ, the precondition of δ is either $id(\delta)$, or its parent should be present (see Figure 2.5 in Chapter 2). Because of the $\textsc{CompDep}$ rule (P1), i.e., $x \in \Pi \Rightarrow \textsc{Parent}(x) \in \Pi$, changes to entities in Π and their parents are kept. Therefore, any change $\delta \in H'$ remains applicable. □

Lemma 2. *(Type Safety). $H'(p_0)$ is well-typed.*

Proof. Entities outside of compilation set stay unchanged, except for method lookup changes (which might be kept and do not affect type soundness); and their referenced targets are preserved since deletions are omitted. Thus, non-compilation set entities remain well-typed. By similar inductive argument as in Lemma 1 and the completeness of the $\textsc{CompDep}$ rules, we have that the compilation set entities also stay well-typed after the slicing. Thus, $H'(p_0)$ is well-typed. □

Theorem 1. *(Correctness of Algorithm 1). Let $\langle p_1, \ldots, p_k\rangle$ be k consecutive subsequent versions of a program p_0 such that $p_i \in P$ and p_i is well-typed for all indices $0 \leq i \leq k$. Let $H = \langle \Delta_1, \ldots, \Delta_k\rangle$ such that $\Delta_i(p_{i-1}) = p_i$ for all indices $1 \leq i \leq k$. Let T be a test suite such that $p_k \models T$. Then the sliced history $H' = \textsc{SemanticSlice}(p_0, H, T)$ is semantics-preserving with respect to T.*

Proof. According to Definition 10, we need to show that H' satisfies the following properties,

1. $H'(p_0) \in P$,
2. $H'(p_0)$ is well-typed,
3. $H'(p_0) \models T$.

From Lemma 1 and Lemma 2 we know that $(H' \circ H_{1..i})(p_0)$ satisfies (1) and (2) is an invariant for the outer loop (Lines 6-17) of Algorithm 1. The original history H has a finite length k, so upon termination, we have $H'(p_0)$ satisfies (1) and (2). Since all functional set insertions and updates are kept in H', any functional set entity that exists in $H(p_0)$ can be found identical in $H'(p_0)$. Because all changes that alter method lookups are also kept (Line 9), the execution traces do not change either. Due to that reason, and by the definition of functional set, (3) also holds. Thus, $H'(p_0)$ satisfies (1), (2) and (3). □

4.3.3 SCM Adaptation

The proposed semantic slicing algorithm operates on the atomic change level and can directly be used with semantic-based tools, such as SemanticMerge. As an optional step, SCM adaptation integrates the generic Algorithm 1 with text-based SCM systems such as Git.

Eliminating Side-Effects

To make the integration with text-based SCM systems easier, each atomic change has to be mapped back to a commit in the original history. The sub-history $H' = \overline{\delta_i} = \langle \Delta'_1, \ldots, \Delta'_k \rangle$ (Δ'_i is possibly empty) returned by SemanticSlice is a sequence of atomic changes labeled by indices indicating their corresponding original commits. A non-empty sliced change set Δ'_i can thus be mapped to its counterpart in the original history, i.e., Δ_i.

However, original commits may contain changes that are sliced away by Algorithm 1. These changes might create unwanted side-effects which break the type safety of the compilation set entities. We deal with this issue by restoring entities that are outside of the compilation set to their original state as in the initial version of the program, thereby "selectively" ignoring these unwanted changes and eliminating the side-effects. We do that by creating an additional commit that reverts unwanted changes on the corresponding code entities.

Calculating Hunk Dependencies

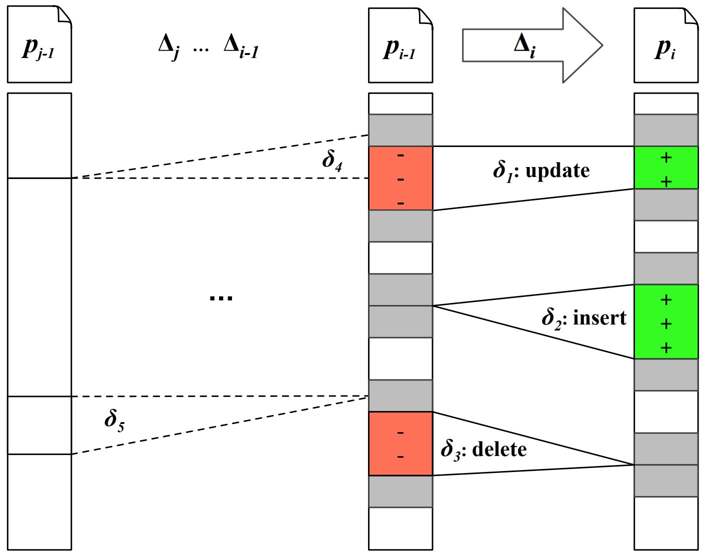


Figure 4.6: Illustration of direct hunk dependencies. Commit Δ_i directly hunk-depends on a previous commit Δ_j. Hunk δ_4 provides contexts for hunk δ_1. Hunk δ_5 provides contexts for hunk δ_3.

Algorithm 1 treats changes between versions as tree edit operations. Another view of changes used by text-based SCM tools is called *hunk* (Section 2.3.2). A hunk is a group of adjacent or nearby line insertions or deletions with surrounding context lines which stay unchanged. For simplicity, we reuse the notations of tree change operations for hunk changes. For example, Figure 4.6 shows an abstract view of the changes made between p_{i-1} and p_i, where blocks with "-" represent lines removed and blocks with "+" represent lines inserted. Grey blocks surrounding the changed lines represent the contexts. From the text-based view, the difference between p_{i-1} and p_i consists of three hunks, i.e., δ_1, δ_2 and δ_3. We define two auxiliary functions, $left(\delta)$ and $right(\delta)$, which return the lines involved before and after the hunk δ, respectively. Special cases are $right(\delta)$ when δ is a deletion and $left(\delta)$ when δ is an insertion: in both cases, the functions return a zero-length placeholder at the appropriate positions.

In order to apply the sliced results with text-based SCM tools where changes are represented as hunks, it is needed to ensure that no conflict arises due to unmatched contexts. Informally, a commit Δ_i *directly hunk-depends* on another commit Δ_j, denoted by $\Delta_i \rightsquigarrow \Delta_j$, if and only if Δ_j *contributes* to the hunks or their contexts in Δ_i. In contrast, if Δ_i does not directly hunk-depend on Δ_j, we say they *commute* [Dar16], i.e., reordering them in history does not cause conflict. The procedure $\textsc{HunkDep}(H')$ returns the transitive hunk dependencies for all commits in H', i.e.,

$$\textsc{HunkDep}(H') \triangleq \bigcup_{\Delta_i \in H'} \{\Delta_j \in H/H' | \Delta_i \rightsquigarrow^* \Delta_j\}.$$

Once a sub-history H' is computed and returned by Algorithm 1, we augment H' with $\textsc{HunkDep}(H')$ and the result is guaranteed to apply to p_0 without edit conflicts.

Algorithm 2 The DirectHunk procedure.

```
1: procedure DirectHunk($B_i, H_{1..i}$)
2:     $D \leftarrow \emptyset$
3:     $B_{i-1} \leftarrow L_i$
4:     for $\delta \in \Delta_{i-1}$ do
5:         if $\delta$ is Del $\wedge$ $right(\delta) \in range(B_i)$ then
6:             $D \leftarrow D \cup \{\Delta_{i-1}\}$
7:         else if $\delta$ is Ins $\wedge$ $right(\delta) \cap B_i \neq \emptyset$ then
8:             $D \leftarrow D \cup \{\Delta_{i-1}\}$
9:             $B_{i-1} \leftarrow B_{i-1} \setminus right(\delta)$
10:    end for
11:    $D \leftarrow D \cup$ DirectHunk($B_{i-1}, H_{1..(i-1)}$)
12:    return $D$
13: end procedure
```

Given a commit Δ_i, we collect a set of text lines B_i which are required as the *basis* for applying Δ_i. For example, B_i for Δ_i includes $left(\delta)$ for all $\delta \in \Delta_i$ and their surrounding contexts (all shaded blocks under p_{i-1} in Figure 4.6). Algorithm 2 describes the algorithm for computing the set of direct hunk dependencies ($\rightsquigarrow$) by tracing back in history and locating the latest commits that contribute to each line of the basis. Starting from Δ_{i-1}, we iterate backwards through all preceding commits. If a commit Δ contains a deletion that falls in the range of the basis (Line 5) or an insertion that adds lines to the basis (Line 7), then Δ is added to the direct dependency set D. In Figure 4.6, $\Delta_i \rightsquigarrow \Delta_j$ because Δ_j has both an insertion (δ_4) and a deletion (δ_5) that directly contribute to the basis at p_{i-1}. When the origin of a line is located in the history, the line is removed from the basis set (Line 9). The algorithm then recursively traces the origin of the remaining lines in B_{i-1}. Upon termination, D contains all direct hunk dependencies of Δ_i. In the worst case, HunkDep calls DirectHunk for every change set in H'. Thus, the running time of HunkDep is bounded above by $O(|H'| \times |H| \times \max_{\Delta \in H}(|\Delta|))$.

4.4 Implementation and Optimizations

In this section, we describe the implementation details of our semantic slicing tool – CSlicer, and discuss practical issues as well as some optimizations applied.

4.4.1 Implementation

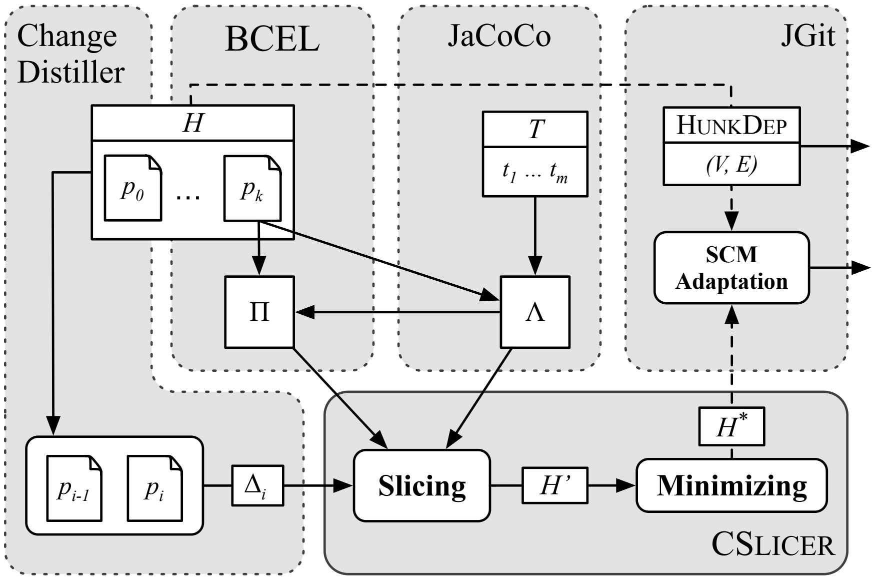


Figure 4.7: Architecture of the CSlicer implementation.

Figure 4.7 shows the high-level architecture of our CSlicer implementation. We implemented CSlicer in Java, using the JaCoCo Java Code Coverage Library [Jac16] for byte code instrumentation and collecting execution data during runtime. We modified the Java source code change extraction tool ChangeDistiller [FWPG07] for AST differencing and change classification. We also used the Apache Byte Code Engineering Library (BCEL) [BCE15] for entity reference relation analysis. The hunk dependency detection component HunkDep was developed based on the Java-based Git implementation, JGit [JGi16]. The HunkDep component can also be used as a stand-alone hunk dependency analysis tool for Git repositories. Given a set of commits, HunkDep generates a hunk dependency graph which visualizes the hunk-level relationship among commits and can be used to reorder commit histories without causing conflicts.

Our CSlicer implementation works with Java projects hosted in Git repositories. The test-slice-verify process is fully-automated for projects built with Maven [Mvn15]. For other build environments, a user is required to manually build and collect test execution data through the JaCoCo plugins. When the analysis is finished, CSlicer automatically cherry-picks the identified commits and verifies the test results. CSlicer can also run in the minimization mode where all cherry-pickable sub-histories are enumerated for further investigation.

The implementation of CSlicer takes about 20 KLOC, and the source code is made available online at: https://bitbucket.org/liyistc/gitslice.

4.4.2 Optimizations and Adaptations

In order to deal with real-life software projects, we implemented a number of techniques and adaptations on top of the CSlicer algorithm which address specific challenges in practice. We describe them below.

Handling Advanced Java Features

We presented our algorithms based on the simplified language P. When dealing with full Java, advanced language features including method overloading, abstract class and exception handling need to be taken into account. For example, various constructs such as `instanceof` and exception `catch` blocks test the runtime type of an object. Therefore, class hierarchy changes may alter runtime behaviors of the test [RST+04]. To address this, we treat class hierarchy changes as an update to the methods that check the corresponding runtime types, to signal possible behavior changes. Changes that may affect method overloading and field overshadowing are detected and included in the sliced history to keep our approach sound. Since reflection related changes are rare in practice, e.g., none of our case studies contained such changes, we disregard them in this work.

Handling Changes in Non-Java Files

Real software version histories often contain changes to non-Java files, e.g., build scripts, configuration files and binary files. Sometimes changes to non-Java files are entangled with Java file changes in the same commits. To avoid false hunk dependencies, we ignore non-Java changes in the analysis and conservatively update all non-Java files to their latest versions unless they are explicitly marked irrelevant by the user. In extremely rare cases, this may cause compilation issues, when older Java components are incompatible with the updated non-Java files. Then the components which cause the problem should be updated or reverted accordingly. None of these affects the target functionalities.

Additional Configurability

We noticed in the experiments that domain knowledge that users have about their projects can enhance the precision of the slicing results. To make the technique more configurable, we allow users to encode their domain knowledge during both the analysis and the cherry-picking processes. Specific packages, files, classes and methods can be included or excluded following user guidance. By default, all changes in the test files, which are not part of the target system, are ignored, as are changes to the internal debugging code.

Another easy-to-implement optimization is ignoring a commit and its revert, if these are found in the history, since removing such a pair does not affect the correctness of the approach.

4.5 Evaluation

In this section, we measure the effectiveness and applicability of CSlicer through both case studies and empirical evaluations. Specifically, we evaluate CSlicer from two different angles:

1. Qualitative assessment of CSlicer in practical settings (Section 4.5.1). We carried out a number of case studies to test the applicability of our techniques in practice, for software maintenance tasks such as porting patches, creating pull request, etc.

2. Quantitative evaluation of CSlicer (Section 4.5.2). We conducted several experiments to get deeper insights on the proposed algorithm to answer questions such as "how much reduction can CSlicer achieve", "how well does CSlicer scale with increasing history length", etc.

4.5.1 Qualitative Assessment of CSlicer

We have conducted three case studies and thoroughly investigated the results produced by CSlicer, to better understand its applicability, effectiveness, and limitations. In particular, we looked at six open-source software projects of varying sizes, different version control workflows, and disparate committing styles – Apache Hadoop [Had15], Elasticsearch [Ela15], Apache Maven [Mvn15], Apache Commons Collection [Col16], Apache Commons Math [Mat16], and Apache Commons IO [IO16]. We chose one target functionality from each project based on the criterion that the functionality should be well-documented and accompanied by deterministic unit tests.

Table 4.1: Statistics of the case study subjects used in the evaluation of CSlicer.

Case	Project	#Files	LOC	$\|H\|$	Changed			$\|T\|$
					f	+	−	
1	Hadoop	5,861	1,291K	267	1,197	111,119	14,064	58
2	Elasticsearch	3,865	616K	51	75	1,755	304	2
3	Maven	967	81K	50	16	1,012	250	7
	Collections	525	62K	39	46	1,678	323	13
	Math	1,410	188K	33	34	1,531	359	1
	IO	227	29K	26	59	975	468	13

Statistics of the subjects are given in Table 4.1 grouped by their case numbers (Column "Case"). Columns "#Files" and "LOC" list the number of Java files and lines of code in the respective projects. Column "$|H|$" lists the number of commits within the chosen history fragment. Column "Changed" shows the number of files changed (f), lines added (+) and lines deleted (−) for the chosen range. Column "$|T|$" shows the number of test cases in the target test suites. For example, the Hadoop project has 6,608 files, out of which 5,861 are Java files. The selected history segment covered changes to 1,197 files where over 100 KLOC were added and about 14 KLOC were deleted.

All of the following studies were conducted on a desktop computer running Linux with an Intel i7 3.4GHz processor and 16GB of RAM. Now we describe each case study in detail.

Experiment 1: Branch Refactoring

We applied CSlicer to Hadoop for *branch refactoring*. The feature "HDFS-6581" was developed in a feature branch (also called a topic branch) which was separated from the main development branch. However, when the development cycle of a feature is long, it is reasonable to merge changes from the main branch back to the feature branch periodically, in order to prevent divergence and resolve conflicts early. And that is exactly the workflow followed by the Hadoop team on their feature branches. As a result, not all commits on the feature branch are logically related to the target feature or required to pass the feature tests. That is, the branch "origin/HDFS-6581" is mixed with both feature commits and merge commits from the main branch. Using CSlicer, we were able to re-group commits according to their semantic functionalities and reconstruct a new feature branch that is fully functional and dedicated to the target feature.

We started with the original feature branch which consists of 42 feature commits and 47 merge commits (each can be expanded into one or more commits from the source branch). There are 34 *auto merges* ("fast-forward merges" in Git terms) which are simply combinations of commits from both branches without conflicts or additional edits. The other 13 are *conflict resolution merges* which contain additional edits to resolve conflicts. To achieve higher granularity when analyzing merge commits, we kept the resolution merges and expanded the auto merges by replaying (cherry-picking) the corresponding commits from the main branch onto the feature branch. Effectively, we converted the branched history into an equivalent expanded linear history by splitting bulk merge commits (see Figure 4.8). This was all done automatically as a preprocessing step. The expanded feature branch has 267 commits in total.

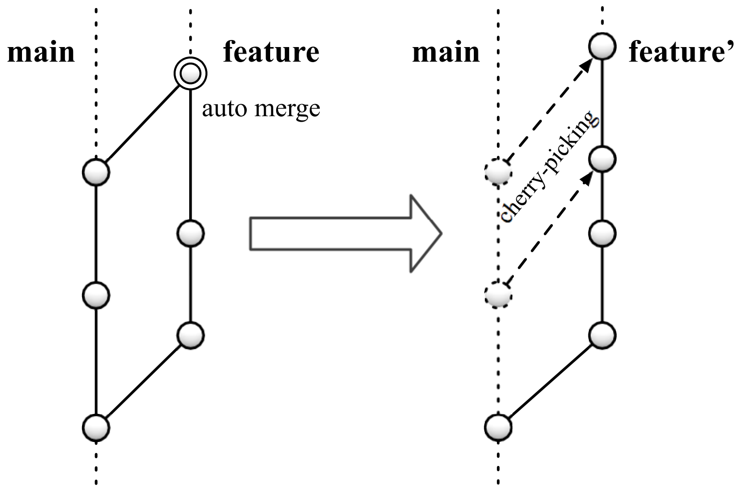


Figure 4.8: Illustration of expanding auto merges through cherry-picking of corresponding commits from the main branch to the feature branch. The auto merge commit is in double circle. The dashed arrows represent cherry-picking of commits. The feature branch after branch refactoring is labeled as feature'.

We executed 58 feature-related unit tests specified in the test plan, which took about 750 seconds to finish. CSlicer identified 65 commits which are required for preserving the test behavior as well as compilation dependencies, and additional 26 commits for hunk dependencies. Note that some commits from the main branch are actually required by the target feature. The refactored feature branch passed the feature test suite and it only contains 91 commits in total, which achieves ∼66% reduction of unnecessary commits.

Experiment 2: Back-porting Commits

The second use case of CSlicer is to identify the set of commits required for back-porting a functionality to earlier versions of a software project. We took a fragment of history between v1.3.6 to v1.3.8 of Elasticsearch and a feature enhancement on the "Groovy interface". There are 2 unit tests clearly marked by the developers in the commit messages intending to test the target functionality introduced in v1.3.8.

As discussed in Section 4.3, there is no efficient algorithm that returns the optimal solution in general. Finding the shortest semantics-preserving sub-history for a given set of tests is a highly challenging task even for programmers with expertise within the software projects. Yet, we manually identified the optimal solutions for this case, which requires 4 out of 51 commits to be ported to v1.3.6 in order for the functionality to work correctly.

CSlicer identified 17 commits achieving a 67% reduction of the unrelated commits. However, compared with the optimal solution, CSlicer reported 13 false positives. We examined all the false alarms and concluded that the main reason causing them is that the actual test execution exposes more behaviors of the system than what were intended to be verified. For instance, the test case "DynamicBlacklist" invoked not only the components implementing the "dynamic black list" but also those that implement the logging functions for debug purposes. Obviously, changes to the logging functions do not affect the test results. But without prior knowledge, CSlicer would conservatively classify them as possibly affecting changes. We investigate common sources causing such false alarms in Section 4.5.3.

Experiment 3: Creating Pull Requests

Another important use case of CSlicer 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. 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.

Table 4.2: Pull requests recreating results in Experiment 3.

Project	ID	Status	$\|H^*\|$	CSlicer		
				$\|H'\|$	$P(\%)$	$R(\%)$
Maven	#74	Open	3	3	100.0	100.0
Collections	#7	Merged	8	5	100.0	62.5
Math	#10	Open	2	2	100.0	100.0
IO	#17	Accepted	9	8	87.5	77.8

To assess the effectiveness of CSlicer in assisting and automating the process of creating pull requests, we selected four public pull requests[1] from four different software projects (see Table 4.2). Columns "ID" and "Status" show the ID and status of the corresponding pull requests. Column "$|H^*|$" shows the length of the pull requests created and submitted by the developers. Columns "$|H'|$", "$P(\%)$", and "$R(\%)$" list the length, precision, and recall of the pull requests created by CSlicer, respectively.

We browsed the pull request lists available from the project repositories and selected the most recent pull requests which are non-trivial (containing more than one commit) and accompanied by unit tests. We used the test cases submitted along with the pull requests as our target tests and used CSlicer to identify the closely related commits from the developers' local histories in their forked repositories.

Two pull requests (#7 in Commons Collections and #17 in Commons IO) have already been finalized (accepted) or merged into the public repositories. The other two are still awaiting approval. For the pull requests #74 in Maven and #10 in Commons Math Library, CSlicer successfully recreated the

[1]The pull requests are subject to future modifications. The pull requests used in this study were taken from the projects' public repositories on Sep 10, 2016.

exact same results as the ones submitted by the developers. For Commons IO, CSlicer included one extra commit (`#bccf2af`) and missed two commits (`#62535cc` and `#3b71609`). For Commons Collections, CSlicer missed three commits.

After a detailed analysis, we discovered a few reasons for CSlicer to miss certain commits. First, several commits in pull request #17 simply reorganize Java import statements, i.e., remove unused imports (`#3b71609`) and replace groups of imports by wild card (`#62535cc`). CSlicer currently ignores all changes to import statements since they do not affect test executions when every version of the program compiles. Second, CSlicer ignores commits which only modify comments and Javadoc. This is currently a limitation of our tool. In fact, accurate identification of changes to relevant documentation is an interesting open research problem. Finally, CSlicer correctly ignores a merge commit (`#4cc49d78`) introducing unnecessary changes in pull request #7 which was included by the developer.

4.5.2 Quantitative Evaluation of CSlicer

To have more insights of the internals of our algorithm, we also empirically evaluated the efficiency and applicability of CSlicer by measuring its performance and the history reduction rate achieved when applied on a benchmark set obtained from real-world software projects. Specifically, we aimed to answer the following research questions:

RQ 1: How effective is the CSlicer *slicing algorithm*, in terms of the number of irrelevant commits identified?

RQ 2: How efficient is CSlicer when applied to *histories of various lengths*?

Experiment 4: CSlicer on Benchmark

Experiment 4 aims to address both RQ 1 and RQ 2 by running CSlicer using the "normal" mode (without minimization).

Subjects and Methodology. In addition to the five projects introduced before, we selected one more project, i.e., Apache ActiveMQ [Act17]. The selected benchmark projects cover a variety of sizes, qualities, objectives, collaboration models, and project management disciplines. For example, Hadoop has the largest code base (∼1,300 KLOC) among the six; Elasticsearch is the most collaborative project which has over 830 contributors; and ActiveMQ has the largest number of sub-projects (36 in total).

From each project, we randomly chose three to four functionalities (e.g., a feature, an enhancement or a bug fix) that are accompanied by good documentation and unit tests. All of the selected projects have, to a certain extent, requirements concerning test adequacy – a patch must be adequately tested before being merged into the repository.

It is also often possible to link specific commits with on-line issue tracking documentation via ticket numbers embedded in the commit messages. For each functionality, we referred to the log messages and ticket numbers to locate the target commit where the functionality was completed and tested. The set of tests are either explicitly mentioned in the accompanied test plan or implicitly enclosed within the same commit as the functionality itself.

The description and target commit for each functionality are shown in Table 4.3. Column "Project" shows the names of the projects where the functionalities were chosen from. Columns "ID", "Type", "Description" and "Commit" represent the identifiers, functionality types (i.e., either feature, bug fix, or

Table 4.3: Target functionalities used in Experiment 4.

Project	ID	Type	Description	Commit	Test Class # Test Methods	$\|T\|$
Hadoop	H1	Feature	*Add NodeLabel operations in help command*	e1ee0d4	TestRMAdminCLI # {testHelp}	1
	H2	Bug Fix	*FileContext.getFileContext can stack overflow if default fs is mis-configured*	b9d4976	TestFileContext # {testDefaultURIWithoutScheme}	1
	H3	Feature	*LazyPersistFiles: add flag persistence, ability to write replicas to RAM disk, lazy writes to disk, etc.*	3f64c4a	TestLazyPersistFiles # {...}	11
	H4	Enhance	*HDFS inotify: add defaultBlockSize to CreateEvent*	6e13fc6	TestDFSInotifyEventInputStream # {testBasic}	1
Elastic	E1	Feature	*Calculate Alder32 Checksums for legacy files in Store#checkIntegrity*	b2621c9	StoreTest # {testCheckIntegrity}	1
	E2	Enhance	*Make groovy sandbox method blacklist dynamically additive*	64d8e2a	GroovySandboxScriptTests # {testDynamicBlacklist}	1
	E3	Feature	*Adding parse gates for valid GeoJSON coordinates*	418de6f	GeoJSONShapeParserTests # {testParse_invalidPolygon}	1
	E4	Enhance	*Enable ClusterInfoService by default*	4683e3c	ClusterInfoServiceTests # {testClusterInfoServiceCollectsInformation}	1
ActiveMQ	A1	Enhance	*Add trace level log to shared file locker keepAlive*	c17b7fd	SharedFileLockerTest # {...}	4
	A2	Bug Fix	*Fix MQTT virtual topic queue restore prefix*	4a8fec4	PahoMQTTTest # {testVirtualTopicQueueRestore}	1
	A3	Enhance	*Only start connection timeout if not already started the rest of the monitoring*	e5a94bf	DuplexNetworkTest # {testStaysUp}	1
Maven	M1	Bug Fix	*ToolchainManagerPrivate.getToolchainsForType() returns toolchains that are not of expected type*	2d0ec94	DefaultToolchainManagerPrivateTest # {...}	3
	M2	Feature	*Add DefaultToolchainsBuilder and ToolchainsBuildingException*	99f763d	DefaultToolchainsBuilderTest # {...}	6
	M3	Bug Fix	*Fix: execution request populate ignores plugin repositories*	d745f8c	DefaultMavenExecutionRequestPopulatorTest # {testPluginRepositoryInjection}	1

	M4	Enhance	*Fail, rather than just warning, on empty entries*	b8dcb08	DefaultModelValidatorTest # {testEmptyModule}	1
IO	I1	Enhance	*Add ByteArrayOutputStream.toInputStream()*	8960862	ByteArrayOutputStreamTestCase # {...}	3
	I2	Enhance	*FieUtils: broken symlink support*	b9d4976	FileUtilsCleanSymlinksTestCase # {testIdentifiesBrokenSymlinkFile}	1
	I3	Bug Fix	*FilenameUtils should handle embedded null bytes*	63cbfe7	FilenameUtilsTestCase # {...}	37
Math	L1	Enhance	*Add estimation types and NaN handling strategies for Percentile*	afff37e	MedianTest # {testAllTechniquesSingleton}	1
	L2	Bug Fix	*Using diagonal matrix to avoid exhausting memory*	b07ecae	PolynomialFitterTest # {testLargeSample}	1

More details can be found at: http://www.cs.toronto.edu/~liyi/tse/data.html.

enhancement), descriptions and commit IDs of each target functionality. Column "Test Class # Test Methods" shows the test class and method names separated by "#", where some of the method names are omitted when they cannot be fit into the space. Column "$|T|$" lists the number of test cases in the corresponding test suites. For example, H1 ("Add nodeLabel operations in help command") is a feature which adds a new label in the help option of the resource manager administration client command line interface of the Hadoop project. There was one test case (TestRMAdminCLI#testHelp) introduced at revision "`#e1ee0d4`" to validate the implementation of this functionality.

The lifetime of a functionality typically spans a period of 1-4 months which corresponds to around 100 commits for a mid-sized project under active development [LZRC16]. In order to evaluate the effectiveness and performance of CSlicer under different contexts, we took three sets of histories of lengths 50 (short), 150 (medium) and 250 (long) tracing back from the target commits. We separated project source code from test code and used CSlicer to perform the semantic slicing on source code only. After applying the sliced histories on top of the base version, we then verified that the resulting programs compile successfully and pass the original tests (precompiled into binaries) selected as the slicing criteria.

Results. The slice compositions for medium-length histories are reported in Figure 4.9. Left y-axis represents percentage of functional (FUNC), compilation (COMP) and hunk (HUNK) dependencies over $|H|$ (%). Right y-axis represents sizes of the functional set ($|\Lambda|$). The history slices returned by CSlicer consist of functional, compilation and hunk dependencies. Each stacked bar in Figure 4.9 represents the percentage of all three types of dependencies within the original histories. The dotted line rising from left to right represents the sizes of functional sets, i.e., the number of source code entities traversed by the test execution. For example, the functional set size for H4 is 4,846. Its sliced history consists of 20.0% functional, 0.7% compilation, and 11.3% hunk dependencies of the original history commits.

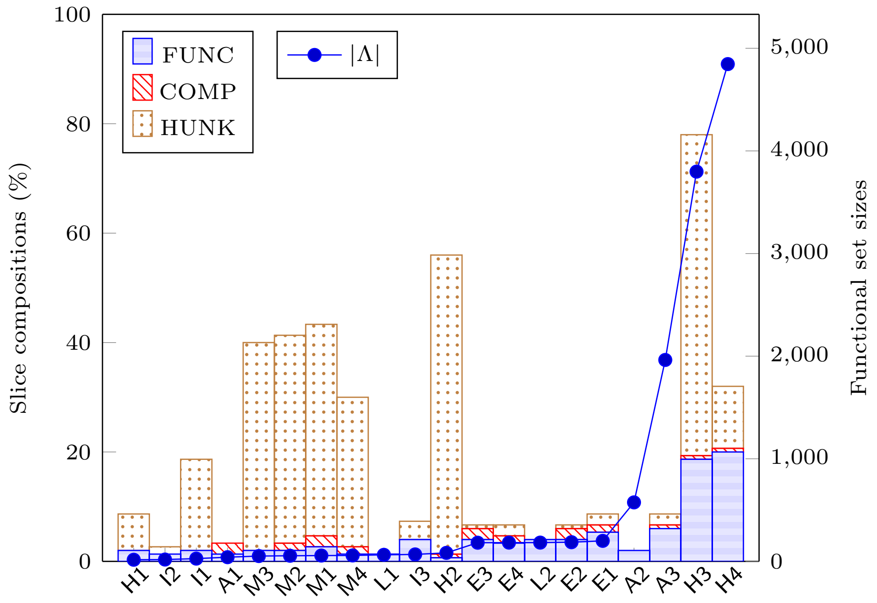


Figure 4.9: Compositions of sliced histories for all benchmarks shown in the order of increasing functional set sizes.

The first observation is that simpler and clear-cut functionalities tend to produce smaller slices. The sizes of functionalities are reflected by the functional set sizes. In general, increasing functional set size leads to increasing size of the history slice (without considering hunk dependencies).

Another interesting observation is that the number of hunk dependencies for Hadoop and Maven is much larger than those of the other projects. The functional set sizes have no obvious relationship with the number of hunk dependencies, which corroborates our conjecture that the level of text-level coupling among commits is project specific.

Answer to RQ 1. CSlicer effectively reduces irrelevant commits given a target test suite.

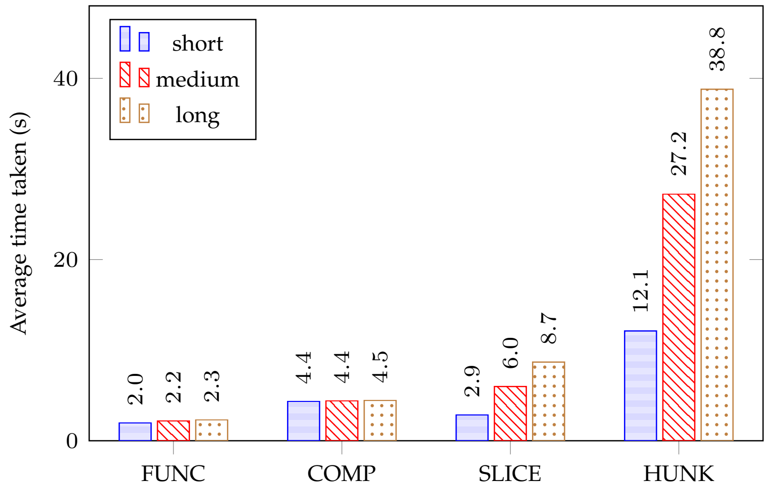


Figure 4.10: Average time taken by CSlicer when running on short, medium and long histories.

Finally, we compare the average time taken by each CSlicer component, i.e., the functional set computation, the compilation set computation, the core slicing component and the hunk dependency computation, when analyzing short, medium and long histories on all the 23 benchmarks. The results are shown in Figure 4.10. Each vertical bar plots the average time taken by the functional set computation (FUNC), compilation set computation (COMP), core slicing component (SLICE) and the hunk dependency computation (HUNK) when running on histories with different lengths. For example, the core slicing component takes 2.9, 6.0, and 8.7 seconds on average to finish for short, medium and long histories, respectively.

Overall, CSlicer takes on average under 10 seconds to finish without computing hunk dependencies. The corresponding minimum and maximum times are 2.6 and 27.0 seconds, respectively. The time spent for FUNC and COMP stays almost constant across different history lengths while the SLICE time grows linearly. The majority of time is spent in computing HUNK which also grows linearly over history length.

Answer to RQ 2. CSlicer scales well on real-world software projects and histories of moderate lengths.

Threats to Validity

Due to limitations of the tool, we only selected software projects which could be configured and built using Maven. The reduction rate, however, depends on many factors – the committing styles, the complexities of the test (how many components it invokes), and coding styles (how closely the components are coupled), etc. While our results are encouraging, we do not have enough data to conclude that they will generalize to all software projects.

As an assumption of our technique, functionalities of interest should be accompanied by appropriate test suites in order to get intended results. Although we were able to identify the corresponding unit tests for all the target functionalities in our selection of the experiment subjects, we understand that this may not always be possible for other, less disciplined, projects. In the absence of well-designed tests, additional expert knowledge might be necessary for refining the slicing results.

4.5.3 Analysis: Sources of Imprecision

Algorithm 1 presented in Section 4.3 assumes that any change on the functional set can potentially alter the final test results and thus all functional changes are kept during slicing. But this assumption is often found to be too conservative in practice. We observed many cases of false positives during change classification in our experiments which can be divided into two groups, namely (1) refactoring changes, and (2) oracle-unobservable changes.

```
1  class Dog {
2    int age;
3    Set<Dog> enemies = new HashSet<Dog>();
4    public Dog (int a) { age = a; }
5    void barking() {
6      System.out.println("bark!");
7    }
8    boolean fighting(Dog other) {
9      barking();
10     enemies.add(other);
11     return !(age<1 || age>5) && age>other.age;
12   }
13 }
14
15 class TestDog {
16   @Test
17   public testFight() {
18     Dog d1 = new Dog(2);
19     Dog d2 = new Dog(1);
20     assertTrue(d1.fighting(d2));
21   }
22 }
```

	Line	Descriptions
δ_1	3	`new HashSet<Dog>()` $\rightarrow$ `new HashSet<>()`
δ_2	6	`bark!` $\rightarrow$ `bark!bark!`
δ_3	11	`!(age<1||age>5)` $\rightarrow$ `age>0&&age<6`

Figure 4.11: Illustration of different types of false positives in Example 2.

Example 2

Figure 4.11 shows an example illustrating the two types of false positives. The `fighting` method of the `Dog` class is tested using two newly created instances. The executed code entities include initialization of the field `enemies`, class constructor, the `barking` and the `fighting` methods. However, none of the changes (δ_1, δ_2, and δ_3) has any influence on the asserted result (Line 20). Specifically, δ_1 is a syntactic rewriting which does not change the semantics of the program at all; δ_2 updates the `barking` method to produce a different console output, but the output is never checked against an oracle; δ_3 changes the returned expression of method `fighting`, but the returned value is not affected at runtime.

Refactoring Changes. *Refactoring changes* [Fow99] are program transformations that change the structure or appearance of a program but not its behavior (e.g., δ_1). Refactoring is important for improving code readability and maintainability. However, refactoring changes create problems for text-based SCM systems, especially during merging. Based on the study by Dig et al. [DMJN08], merging changes along with code refactoring causes significantly more merge conflicts, compilation and runtime errors. The common practice is thus separating refactoring from functional changes [DMJN08] which gives developers the flexibility to replay the refactoring changes after merging is done. Therefore, if we can successfully isolate refactoring changes from other changes, then we will be able to produce logically clean and easy-to-merge history slices.

Renaming code entities is one type of commonly seen refactoring change. In fact, existing code refactoring detection techniques [DCMJ06, GW05, KR11] focus on renaming and movement of structural nodes, i.e., packages, classes, methods and fields. However, such changes would alter AST structures as well as node identifiers and thus often have a repercussion on later changes. For instance, once a class renaming change is applied, all successive references to that class have to use the updated name. To preserve correctness of Algorithm 1, we should only consider self-contained local and low-level refactorings [MHPB12] as candidates to be dropped. For example, one common change pattern that can be ignored is the usage of syntactic sugars and updated language features. In the Apache Maven [Mvn15] change history, we observe the adoption of new Java 7 features `try-with-resources` statement [Try16] and the diamond operator [Dia16] in a massive scale.

Oracle-Unobservable Changes. Execution results of a test suite depend on both the *explicit* and *implicit* checks embedded in the tests. In the case of Java, a JUnit [JUn16] test case fails either due to assertion failures (explicit checks defined by developers) or runtime errors (implicit checks performed by runtime system) [SZ11]. Some changes, even if may alter the program behaviors, are unobservable to the test oracle and therefore they are referred to as *oracle-unobservable changes*. The reason is that the updated behaviors are not checked either explicitly or implicitly and therefore would not affect test results in any way (e.g., δ_2 and δ_3). Algorithm 1 does not distinguish such changes from the ones that do affect test results, and this may cause inclusion of undesired changes.

Yet, in some cases, oracle-unobservable changes may be desirable and should be kept in the history slice, even if the given tests are inadequate as slicing criteria to distinguish the target semantic functionalities. For example, performance enhancement changes which only improve the execution efficiency and do not affect the functional aspect of the program, are unobservable with respect to pure functional tests. Performance tests [Mol09] can be used instead to capture the performance aspect of the programs. Algorithm 1 is conservative in design and always includes oracle-unobservable changes in the slicing results.

4.5.4 Evaluation Summary

To summarize, we analyzed CSlicer both qualitatively and quantitatively. We demonstrated that apart from assisting developers in porting functionalities, CSlicer can also be applied for other maintenance tasks such as branch refactoring and creating logically focused pull requests. Furthermore, the results of quantitative studies suggest that CSlicer is able to achieve good reduction of irrelevant commits at a relatively low cost when applied to real-world software repositories, which justifies its value in practice.

4.6 Related Work

To the best of our knowledge, the software history semantic slicing problem was first defined in our prior work [LRC15]. However, the approach of finding semantic slices using change dependency analysis does intersect with many areas spanning history understanding and manipulation, code change classification, change impact analysis (reviewed in Section 1.2) and software product line variants generation. We compare CSlicer with the related work below.

4.6.1 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 [MHPB12, BHEN13], localize bugs [Zel99, ZH02], and support predictions [ZWDZ04, HZ13].

An interesting take on history analysis is *history transformation* [HOZ+12, MSBE15]. Muşlu et al. [MSBE15] introduced a concept of *multi-grained* development history views. Instead of using a fixed representation of the change history, they propose a more flexible framework which can transform version histories into different representations at various levels of granularity to better facilitate the tasks at hand. Such transformation operators can be combined with CSlicer to build a history view which clusters semantically related changes as high-level logical groups. This semantics summarization view [MSBE15] is a much more meaningful representation for history understanding and analysis.

4.6.2 Change Classification

The CSlicer algorithm relies on sophisticated *structural differencing* [CRGW96, FMB+14, Bil05] and *code change classification* [FG06, FWPG07, HM08] algorithms. We use the former to compute an optimal sequence of atomic edit operations that can transform one AST into another, and the latter to classify atomic changes according to their change types.

The most established AST differencing algorithm is ChangeDistiller [FWPG07]. It uses individual statements as the smallest AST nodes and categorizes source code changes into four types of elementary tree edit operations, namely, insert, delete, move and update. We use a slightly different AST model in which all entity nodes are unordered. For example, the ordering of methods in a class does not matter while the ordering of statements in a method does. Hence, the move operation is no longer needed and thus never reported by CSlicer. We also label each AST node using a unique identifier to represent the fully qualified name of each source code entity. The rename of an entity is thus treated as a deletion followed by an insertion. This modification helps avoid confusion in functional set matching using identifiers. Finally, deletion is only defined over leaf nodes in ChangeDistiller. In contrast, we lift this constraint and allow deletion of a subtree to gain more flexibility and ensure integrity of the resulting AST.

4.6.3 Product Line Variant Generation

The *software product line* (SPL) [CN01, PBL05] community faces similar challenges in ensuring well-formedness of programs. An SPL is an efficient means for generating a family of program variants from a common code base [KA08, KAT+09]. Code fragments can be disabled or enabled based on product configurations, for example, using "`#ifdef`" statements in C's preprocessor. However, out of the large number of variants that can be generated, many are ill-formed due to syntax and type errors. Therefore, variant generation algorithms need to check the implementation of SPL to ensure that the generated variants are well-formed.

Kästner et al. [KAT+09] introduced two basic rules for enforcing syntactic correctness of product variants, namely, *optional-only* and *subtree*. The optional-only rule prevents removal of essential language constructs such as class name and only allows optional entities, e.g., methods or fields, to be removed. The subtree rule requires that when an AST node is removed, all of its children are removed as well, Our field- and method-level AST model and the syntactic correctness assumption over intermediate versions together automatically guarantee the satisfaction of the two rules.

Kästner and Apel [KA08] proposed an extended calculus for reasoning about the type-soundness of product line variant programs written in Featherweight Java [IPW01]. They formally proved that their annotation rules on SPL are complete. Our CompDep reference relation rules are directly inspired by theirs. We are, however, able to discard some of the rules since we only deal with code entities at the field- and method-level granularity.

Despite the similarities in syntactic and type safety requirements on the final products, the inputs for both problems differ. Unlike the SPL variant generation problem where a single static artifact is given, the semantic slicing algorithm needs to process *a sequence* of related yet distinct artifacts under evolution. And on top of low-level requirements on program well-formedness, semantic slices also need to satisfy high-level semantic requirements, i.e., some functionality as captured by test behaviors.

4.7 Summary

In this chapter, we proposed CSlicer, an efficient semantic slicing algorithm which resides on top of existing SCM systems. Given a functionality exercised and verified by a set of test cases, CSlicer is able to identify a subset of the history commits such that applying them results in a syntactically correct and well-typed program. The computed semantic slice also captures the interested functional behavior which guarantees the test to pass.

We have also implemented a novel hunk dependency algorithm which fills the gap between language semantic entities and text-based modifications. We identified a number of sources that can cause imprecision in the slicing process. We carried out several case studies and empirically evaluated our prototype implementation on a large number of benchmarks obtained from open-source software projects. We conclude that CSlicer is effective and scales in practical applications.

Chapter 5

Dynamic History Slicing Through Delta Refinement

"History doesn't repeat itself, but it does rhyme."

— Mark Twain

5.1 Introduction

In Chapter 4, we presented CSlicer which analyzes the latest program version to collect test coverage information and then computes an over-approximated set of commits that include changes to the exercised code elements. CSlicer trades accuracy for efficiency: it executes the test suite only once, but it conservatively assumes that all changes traversed by the test execution can potentially alter the test results. This assumption results in potential inclusion of unnecessary or irrelevant changes into the history slice (see Section 4.5.3).

On the other hand, *Git-bisect* [Git16a] and *delta debugging* [ZH02] isolate failure-inducing changes in version histories through divide-and-conquer, where a set of commits is partitioned and tested separately until a minimal subset that exposes the test failures is found. Although solving a slightly different problem than CSlicer, their approaches can also be applied to semantic history slicing and they have better guarantees in result accuracy. Yet, they can be very expensive to run as they execute the test suite multiple times, depending on the way history is partitioned and on the order in which partitions are tested.

The biggest challenge for precisely solving the semantic slicing problem lies in the very large number of possible programs in the search space, i.e., those that are induced by all subsets of commits in the history. A valid semantic history slice has to be efficiently discovered from the exponential number of candidates.

Dynamic Delta Refinement

In this chapter, we propose a precise semantic history slicing technique based on *iterative refinement* and *change significance ranking.* We discover relevance of changes to the target tests through successive test runs and utilize this information to guide the history partition and speed up the refinement process. We refer to this technique as *dynamic delta refinement*. Its key insight is that by comparing the runtime executions of two program versions – before and after a change, we can extract information about the precise impact of the changes at various program points. By combining impact information with test outcomes (pass or fail), we are able to accurately infer the significance of changes with respect to the target tests. In particular, if the tests still pass after removal of a change, then the removed change and its family of related changes are insignificant to the tested functionalities. We give more details on how such families of changes can be detected using dynamic program invariants generated by Daikon [EPG+07] in Section 5.3.

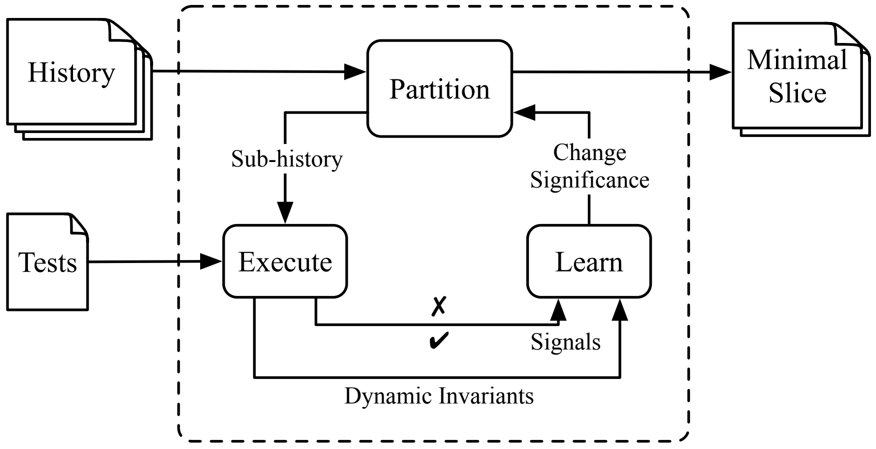


Figure 5.1: Overview of the dynamic delta refinement loop.

Figure 5.1 shows an overview of the delta refinement loop. Using the significance measurements of changes, dynamic delta refinement is able to efficiently find minimal semantic history slices through well informed partition schemes. With much higher confidence, changes of less significance are removed first and, upon success, the analysis scope is reduced and the refinement continues recursively. The results of test executions, either success or failure, are used to update significance ranking of the remaining changes. The ranking accuracy is improved with each execution, and the refinement loop terminates when the minimality condition is met. Note that the algorithm maintains a valid semantic slice throughout this process, so it can be interrupted at any time and return a valid best-effort result.

Contributions

We summarize this chapter's contributions as follows:

- We show how dynamic delta refinement can learn significance of changes with respect to a specific high-level functionality.
- We define minimal semantic slices and describe an algorithm for computing them based on dynamic delta refinement.
- We report on an implementation of a fully-automated precise semantic history slicing tool that operates on Java projects hosted in Git repositories.

- We compare our technique with previous work in terms of precision and efficiency. Based on empirical evaluation, our technique achieves ∼46% precision improvement in terms of the number of commits identified, compared with CSLICER. We also demonstrate the advantage of using change significance to speed up the basic partition scheme used by delta debugging.

Organization

This chapter is organized as follows:

- In Section 5.2, we use two examples to illustrate how dynamic invariants are used for learning change significance and how significance ranking is used to guide history partition.
- In Section 5.3, we formalize the delta refinement algorithm for finding minimal semantic slices and prove its correctness.
- In Section 5.4, we describe our implementation details.
- In Section 5.5, we discuss evaluation setups and experimental results.
- Finally, we discuss related work and conclude in Sections 5.6 and 5.7, respectively.

5.2 Definer by Examples

In this section, we illustrate our approach on two simple examples.

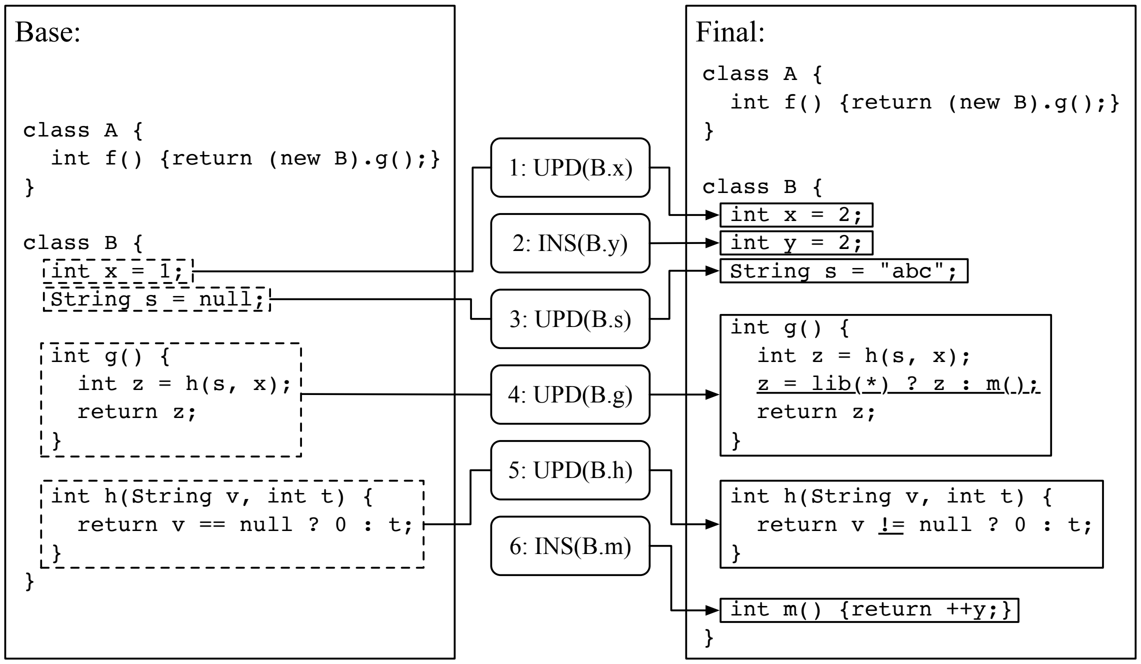
5.2.1 Changes and Dependencies

Example 3

Figure 5.2a shows two versions of a Java program `Foo.java`: "base" and "final". The "final" version introduces a few modifications to the class `B` through a series of *atomic changes*. Atomic changes are defined over the *abstract syntax trees* (ASTs) of the program as *insertions* (INS), *deletions* (DEL), or *updates* (UPD) of AST nodes (e.g., fields, methods, etc.) Specifically, there are six atomic changes between the "base" and the "final" versions (listed in no particular order), $\boxed{1}$: an update to the field `B.x`; $\boxed{2}$: an insertion of a new field `y` into the class `B`; $\boxed{3}$: an update to the field `B.s`; $\boxed{4}$: an update to the method `B.g()`, which adds an additional statement "`z = lib(*) ? z : m()`", conditionally assigning the returned value of `m()` to the local variable `z`; $\boxed{5}$: an update to method `B.h()`, which replaces "`==`" by "`!=`"; and $\boxed{6}$: an insertion of a new method `m()` into the class `B`.

The `lib(*)` method called in `g()` represents an external library whose returned value is only known during runtime. We do know that the library method behaves deterministically across multiple executions, but cannot predict its return value without executing it. The desired functionality of the program is captured by a unit test for `Foo.java` which asserts that the returned value of the method `A.f()` should be equal to 3 (see Figure 5.2b). We denote this test by T. Note that the test assertion holds in the final version of the program but fails in the base one.

A semantic history slice is a subset of the changes which produces a well-formed and fully functional program that can still pass the test. Since we only care about a subset of the program behaviors captured by the test, some atomic changes are unnecessary. In Example 3, the minimal set of changes which



Base:

```
class A {
  int f() {return (new B).g();}
}

class B {
  int x = 1;
  String s = null;

  int g() {
    int z = h(s, x);
    return z;
  }

  int h(String v, int t) {
    return v == null ? 0 : t;
  }
}
```

Final:

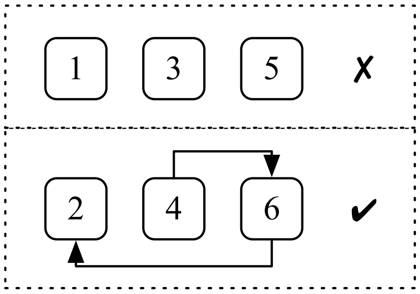
```
class A {
  int f() {return (new B).g();}
}

class B {
  int x = 2;
  int y = 2;
  String s = "abc";

  int g() {
    int z = h(s, x);
    z = lib(*) ? z : m();
    return z;
  }

  int h(String v, int t) {
    return v != null ? 0 : t;
  }

  int m() {return ++y;}
}
```

(a) Atomic changes.

Test:

```
class TestFoo {
  @Test
  void test() {
    A a = new A();
    assertEquals(3, a.f());
  }
}
```

(b) Target test.



(c) Atomic changes between the "base" and "final" versions of `Foo.java`.

Figure 5.2: Illustration of changes and their dependencies.

qualifies as *a valid semantic slice* is $\{\boxed{2}, \boxed{4}, \boxed{6}\}$. The test T fails when any of these three changes is missing and passes whenever all of them are present. Other changes are either never executed or do not alter the asserted values. Interestingly, the test passing property is not monotone in general, i.e., adding modifications to a valid semantic history slice may change the tests from passing to failing. This makes computing *minimal* semantic history slices rather challenging.

Change Dependencies

Atomic changes are not completely independent from each other. In order to construct a well-formed program, some changes have to be applied as prerequisites for others [RST+04, LRC15]. For example, Ins(`B.m`) depends on Ins(`B.y`) since the method `B.m()` accesses the field `B.y` and requires the declaration of the field in order to compile; and Upd(`B.g`) depends on Ins(`B.m`) since the new version of the method `B.g()` invokes `B.m()` (see Figure 5.2c).

We are only interested in producing well-formed programs. The partition of changes thus has to obey the dependency relations. That is, reverting a subset of atomic changes results in a well-formed program only if the remaining changes have all their dependencies satisfied. The change dependencies can be computed systematically, as we describe in Section 5.4.

5.2.2 Learning Change Significance

We now show how delta information observed from successive test runs can be used to learn significance of atomic changes with respect to a target test suite.

In Example 3, the target test T passes in the final version. We can use this information to establish facts about the program variables at various program points by generating *dynamic invariants* [EPG+07]: likely invariants that may not generalize but that hold for the exercised test executions. For simplicity, we refer to them as invariants from now on. For instance, "`B::x == 2`" is a field invariant which indicates that the value of the field `B.x` equals to 2 during the initial execution. Another example is "`A.f()::return == 3`" which is a method post-condition asserting that the return value of `A.f()` is 3.

We denote by H^- the set of reverted changes, I the initial set of invariants for the final version of the program, and I' the invariants after changes are reverted. The row H^- of Figure 5.3 shows four possible cases of reverted changes in Example 3. The deltas in the generated invariants before and after reverting changes is shown in row "$I \setminus I'$" of the table. The rows "$T(H^+)$" and "Signals" show the test outcomes and significance signals learned for each case, respectively. We discuss each case in turn below.

	Case 1	Case 2	Case 3	Case 4
H^-	$\boxed{1}$	$\boxed{1}$ $\boxed{3}$	$\boxed{4}$	$\boxed{3}$ $\boxed{4}$

$I \setminus I'$	`B::x == 2`	`B::x == 2`
		`B::s != null`
		`B.h(I,S)::return == 0`
			`B::y one of {2, 3}`
			`B.g()::return == 3`
			`B.m()::return == 3`
			`A.f()::return == 3`
				`B::s != null`
				`B.h(I,S)::return == 0`
				`B::y one of {2, 3}`
				`B.g()::return == 3`
				`B.m()::return == 3`
				`A.f()::return == 3`

$T(H^+)$	✔	✔	✘	✘
Signals	$\boxed{1}\downarrow$	$\boxed{1}\downarrow\boxed{3}\downarrow\boxed{5}\downarrow$	$\boxed{2}\uparrow\boxed{4}\uparrow\boxed{6}\uparrow$	

Figure 5.3: Change significance learning case by case.

Case 1: Test Passing without Extra Signal

Suppose a singleton atomic change set $H^- = \{\boxed{1}\}$ is reverted during the first partition step, and the new program is now equivalent to applying $H^+ = \{\boxed{2},\boxed{3},\boxed{4},\boxed{5},\boxed{6}\}$ to the base version. The declarations and initializations of `x` are reverted to the base version: `x` initialized to 1 instead of 2.

Static analysis is unable to determine whether this change would affect the test results due to the undetermined returns of `lib(*)`. But we are able to precisely detect the *impact* of reverting $\boxed{1}$ by comparing the new set of invariants I' generated during the actual execution of the new program to the original invariants I. In this case, we observe that only one invariant disappears after reverting $\boxed{1}$, namely, "`B::x == 2`" (see row "$I \setminus I'$" in Figure 5.3). This likely shows that the impact of reverting $\boxed{1}$ is local to the change itself and does not flow into other program points.

In fact, `lib(*)` returns `false` at runtime and thus the change on `B.x` does not propagate through – the returned value from `h(s,x)` is overwritten by `m()` which is independent of the change. The test outcome is unchanged and therefore the value of `B.x` is considered insignificant. We decrease the significance score of $\boxed{1}$ (denoted by $\downarrow$ in Figure 5.3 row "Signals").

Case 2: Test Passing with Extra Signals

Now suppose that two atomic changes, $H^- = \{\boxed{1}, \boxed{3}\}$, are reverted together. The initial values of both `x` and `s` are affected: `x` taking value 1 instead of 2 and `s` being initialized to `null` instead of `"abc"`. This time, we observe three invariants disappearing after the revert: "`B::x == 2`", "`B::s != null`" and "`B.h(I,S)::return == 0`", which involve an additional method `h(I,S)` whose return value is affected by the revert.

Since the test passes again, none of the three invariants in $I \setminus I'$ is consequential for the target functionality. These include the return of `B.h(I,S)`. Apart from $\boxed{1}$ and $\boxed{3}$ which are obviously insignificant, we could also infer that $\boxed{5}$ is insignificant – a local impact analysis shows that $\boxed{5}$ only affects the return of `B.h(I,S)`, At this point, we have determined that the change set $\{\boxed{1}, \boxed{3}, \boxed{5}\}$ is insignificant for the target test. This information can be used to guide the partition in the next iteration by prioritizing reverting $\boxed{5}$ over the rest.

Case 3: Test Failing by Determined Causes

When $\boxed{4}$ is reverted, the conditional assignment statement "`z = lib(*) ?  z :  m()`" in `g()` is removed. The test fails because now the value from `h(s,x)` flows through, which is different from the old value from `m()`. Since an atomic change is already the smallest unit in our analysis, we can pinpoint $\boxed{4}$ as the definite cause of the test failure.

All invariants violated by the revert are directly impacted by the change and most likely cause the failure. They are as follows: "`B::y one of {2, 3}`" which asserts that the field `y` used to take both 2 and 3 (now `y` can only be 2), and "`B.m()::return == 3`" which asserts that the return of `m()` used to be 3 (now `m()` does not return at all). Therefore, we consider both $\boxed{2}$ and $\boxed{6}$, which are associated with `B.y` and `B.m()`, respectively, significant for the test.

Case 4: Test Failing by Undetermined Causes

When the test fails after reverting multiple atomic changes, as happens in this case, the causes for the failure are undetermined. The actual cause can be any one in the reverted changes or their arbitrary combination. For example, when $H^- = \{\boxed{3}, \boxed{4}\}$ is reverted, the test fails only due to the absence of $\boxed{3}$ but we cannot infer useful significance information in this case.

5.2.3 History Partition by Significance Ranking

The basic idea of history partition is inspired by *delta debugging* [ZH02]. In the first iteration, the history is split into two halves which are then tested individually. If one of the partitions passes the test, then the process continues recursively on the successful partition. Otherwise, less aggressive partitions are produced by reverting fewer changes and keeping more. For example, we can split the history into four similar-size change sets and revert each of them, one at a time. If none of the attempts are successful, then finer-grained partitions are produced until we reach a point where only a single atomic change is reverted at a time. Then we are able to classify the change precisely according to the test results. The process terminates when a *1-minimal* [ZH02] history slice is found: reverting any single change in the slice fails the test.

n	Partition (H^+, H^-)	T	Signals
1	1 2 3 4 $\boxed{5}$ $\boxed{6}$ $\boxed{7}$ $\boxed{8}$	-	
	$\boxed{1}$ $\boxed{2}$ $\boxed{3}$ $\boxed{4}$ 5 6 7 8	-	
2	1 2 $\boxed{3}$ $\boxed{4}$ $\boxed{5}$ $\boxed{6}$ $\boxed{7}$ $\boxed{8}$	-	
	$\boxed{1}$ $\boxed{2}$ 3 4 $\boxed{5}$ $\boxed{6}$ $\boxed{7}$ $\boxed{8}$	✘	
	$\boxed{1}$ $\boxed{2}$ $\boxed{3}$ $\boxed{4}$ 5 6 $\boxed{7}$ $\boxed{8}$	-	
	$\boxed{1}$ $\boxed{2}$ $\boxed{3}$ $\boxed{4}$ $\boxed{5}$ $\boxed{6}$ 7 8	✔	$\boxed{3}\downarrow\boxed{5}\downarrow\boxed{7}\downarrow\boxed{8}\downarrow$
3	$\boxed{1}$ $\boxed{2}$ 3 $\boxed{4}$ 5 $\boxed{6}$	✔	$\boxed{3}\downarrow\boxed{5}\downarrow$
4	1 2 $\boxed{4}$ $\boxed{6}$	-	
	$\boxed{1}$ $\boxed{2}$ 4 6	✘	
5	1 $\boxed{2}$ $\boxed{4}$ $\boxed{6}$	✔	$\boxed{1}\downarrow$
6	2 $\boxed{4}$ $\boxed{6}$	-	
	$\boxed{2}$ 4 $\boxed{6}$	✘	$\boxed{2}\uparrow\boxed{4}\uparrow\boxed{6}\uparrow$
	$\boxed{2}$ $\boxed{4}$ 6	-	

Figure 5.4: Enhanced history partition scheme.

In our technique, we make two enhancements to the basic partition scheme:

- before attempting basic partitions, we prioritize removal of low significance changes whenever possible, and
- by precisely analyzing dependencies between changes, we predict compilation errors without needing to compile the program.

Example 4

We use another example with a slightly more complex change history to illustrate our enhanced history partition scheme. In this example, there are eight atomic changes $\{\boxed{1}, \ldots, \boxed{8}\}$, adding two non-essential changes $\boxed{7}$ and $\boxed{8}$ on top of the history in Example 3. The set of essential changes is still $\{\boxed{2}, \boxed{4}, \boxed{6}\}$. The code and detailed changes for Example 4 can be found at: `https://bitbucket.org/liyistc/gitslice/wiki/ase16-e2`.

The actual steps taken when analyzing this example are shown in Figure 5.4. During the first step ($n = 1$), the history is partitioned into two equal halves, i.e., $\{\boxed{1}, \boxed{2}, \boxed{3}, \boxed{4}\}$ and $\{\boxed{5}, \boxed{6}, \boxed{7}, \boxed{8}\}$. We keep one set and revert the other but only to find that the dependencies $\boxed{6} \rightarrow \boxed{2}$ and $\boxed{4} \rightarrow \boxed{6}$ are violated. In Figure 5.4, change dependency violations are represented by "-" in column "T". No test run is needed so far.

During the second step ($n = 2$), we increase the partition granularity and revert two changes at a time. Reverting {$\boxed{3}$,$\boxed{4}$} produces a well-formed program, but the test fails (✘) since $\boxed{4}$ is an essential change. No significance signal is learned since the cause of the failure is not determined. The test passes (✔) when {$\boxed{7}$,$\boxed{8}$} is reverted. The extra signals for $\boxed{3}$ and $\boxed{5}$ that we learn from the passing test allow us to lower their significance as well.

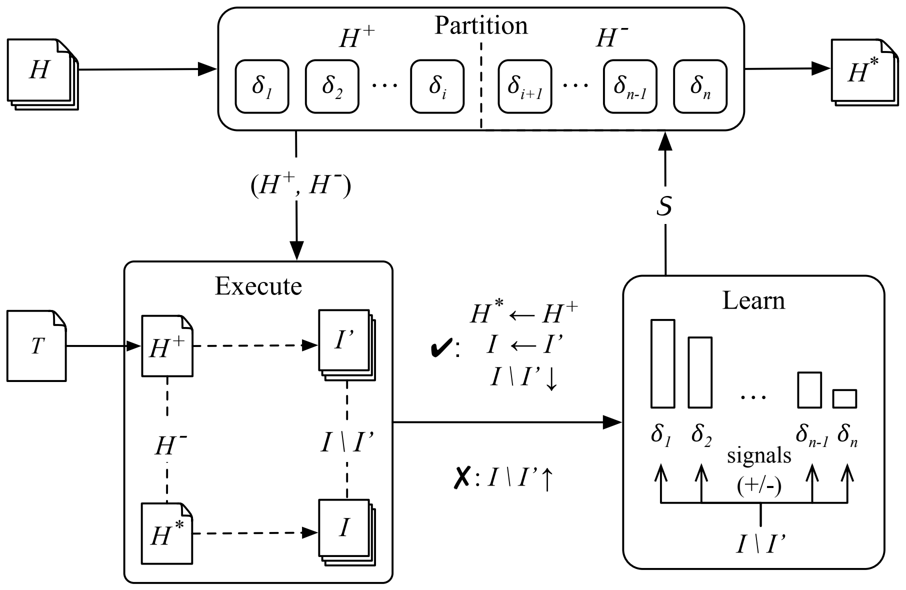
During the third step ($n = 3$), we revert {$\boxed{3}$,$\boxed{5}$} as suggested by their significance measurements and successfully reduce the scope down to only four atomic changes. Similarly to the first step, neither half of the partition produced during Step 4 is a valid semantic slice. Therefore, we increase the partition granularity again in Step 5, reverting a single change at a time. This time, $\boxed{1}$ can be reverted which leaves a valid 1-minimal history semantic slice {$\boxed{2}$,$\boxed{4}$,$\boxed{6}$}. During the final step ($n = 6$), the delta refinement loop terminates because none of the changes can be successfully reverted.

For this example, six test runs are needed for finding the minimal solution using the enhanced partition scheme. In contrast, the basic partition scheme from [ZH02], without significance learning or change dependency analysis, requires thirteen test runs and twelve additional (failed) compilations.

5.3 The Dynamic Delta Refinement Algorithm

In this section, we present the dynamic delta refinement algorithm for precise semantic history slicing in detail.

5.3.1 Algorithm Descriptions



H
Partition
H^+
H^-
δ_1 δ_2 … δ_i δ_{i+1} … δ_{n-1} δ_n
H^*
(H^+, H^-)
S
T
Execute
H^+
I'
H^-
$I \setminus I'$
H^*
I
$H^* \leftarrow H^+$
✔: $I \leftarrow I'$
$I \setminus I' \downarrow$
✘: $I \setminus I' \uparrow$
Learn
δ_1 δ_2 … δ_{n-1} δ_n
signals (+/-)
$I \setminus I'$

Figure 5.5: Dynamic delta refinement overview.

Given a history H and a test suite T, to compute a 1-minimal semantic slice H^*, our algorithm iteratively goes through three phases: *partition*, *execution* and *learning*, as shown in Figure 5.5. To implement each phase, the delta refinement algorithm maintains three data structures:

- H^*, the current minimal semantics-preserving history slice, which is always an over-approximation of the 1-minimal solution and can be returned as a sub-optimal solution if the refinement process terminates prematurely;
- I, the set of dynamic invariants generated from the last successful test execution and updated after every successful run;
- $\mathcal{S} : \Delta \to \mathbb{R}$, the change significance ranking – a function from atomic changes to real numbers, updated according to the outcomes from the execution phase.

Algorithm 3 presents the dynamic delta refinement algorithm as a set of generic rules specifying the minimal requirements for each phase.

Initialization

The Init rule executes tests on the final version $H(p_0)$ and collects dynamic invariants $I = \textsc{Inv}(H, T)$. It also initializes H^* to be the input history H, and initializes significance scores for all atomic changes in H to zero.

Algorithm 3 The dynamic delta refinement algorithm.

Initialization:

$$\frac{}{H^* \leftarrow H \quad \forall \delta \in H \cdot \mathcal{S}(\delta) \leftarrow 0 \quad I \leftarrow \textsc{Inv}(H,T)}\ \textsc{Init}(H,T)$$

Partition:

$$\frac{|H^*| > 1}{\begin{array}{c}H^+ \neq \emptyset \quad H^- \neq \emptyset \quad H^+ \cup H^- = H^* \\ H^+ \cap H^- = \emptyset\end{array}}\ \textsc{Par-Rand}(H^*)$$

$$\frac{\exists \delta_i, \delta_j \in H^* \cdot \mathcal{S}(\delta_i) > \mathcal{S}(\delta_j)}{H^+ \leftarrow \bigcup_{\mathcal{S}(\delta) \geq \mathcal{S}(\delta_i)} \delta \quad H^- \leftarrow H^* \setminus H^+}\ \textsc{Par-Sig}(H^*, \mathcal{S})$$

Execution and Learning:

$$\frac{H^+ \models T \quad I' = \textsc{Inv}(H^+, T)}{(I \setminus I') \downarrow \quad H^* \leftarrow H^+ \quad I \leftarrow I'}\ \textsc{Pass}((H^+, H^-), T)$$

$$\frac{H^+ \not\models T \quad I' = \textsc{Inv}(H^+, T) \quad |H^-| = 1}{(I \setminus I') \uparrow \quad H^* \leftarrow H^* \quad I \leftarrow I}\ \textsc{Fail-1}((H^+, H^-), T)$$

$$\frac{H^+ \not\models T \quad |H^-| > 1}{H^* \leftarrow H^* \quad I \leftarrow I}\ \textsc{Fail-2}((H^+, H^-), T)$$

Partition

This phase receives a history H and splits it into two non-empty sub-histories, H^+ and H^-. The split can be either random or guided by a significance ranking of atomic changes. The two rules for this phase, Par-Rand and Par-Sig, govern the behaviors of two different partition schemes.

The *random partition* splits the current minimal semantic slice H^* into two non-empty sub-histories, H^+ and H^-, randomly, when the length of H^* is greater than one. Then H^+ is kept while H^- is reverted. We adjust the relative sizes of H^+ and H^- to balance between progressions upon test success and chances of successes. For example, a smaller H^+ can reduce a larger chunk of non-essential changes if the tests pass, but it usually has a lower chance of success assuming essential changes are uniformly distributed. In the actual implementation, we gradually increase the size of H^+ when test fails and decrease it otherwise.

The *significance-guided* partition scheme splits the history according to significance ranking of changes, such that all changes in H^+ have higher or equal significance score than those in H^-. With accurate significance ranking, reverting non-essential changes can be very effective. In practice, we apply Par-Sig first whenever possible, as it has a higher chance to produce more accurate splits.

Execution

The execution phase receives a valid partition (H^+, H^-) and executes tests T on $H^+(p_0)$ (written as H^+ afterwards). The dynamic invariants I' generated from the execution are compared with I which is generated from the last successful test run. An invariant delta $I \setminus I'$ and a test signal (✔ / ✘) are passed on to the learning phase.

Learning

The learning phase infers significance of individual atomic changes according to the invariant deltas and the test signals. There are three rules for this phase: Pass, Fail-1 and Fail-2 controlling how the significance ranking is updated under different circumstances.

When H^+ passes T, the Pass rule applies. The invariant deltas indicate differences observed during the current program execution compared with the last successful test run. We use the invariant deltas to find atomic changes which are potentially related to the observed differences in test execution. This step matching variables and program points in invariant deltas with other atomic changes is performed using a simple local static change impact analysis [Arn96]. For each affected method postcondition, we collect all statements within the method body that have potential impacts on the method return (e.g., changed value flows into the return). For instance, using a simple backward data-flow analysis, the invariant "`B.g()::return == 3`" in Example 3 is matched to $\boxed{4}$ which directly updates the returned variable `z`. Similarly, for each method precondition, we consider every call site and collect statements preceding the method invocation which potentially impacts the corresponding input parameters. Finally, for invariants on fields, we analyze all field access sites and perform a similar backward analysis. The significance of each matched atomic change is decreased.

Note that the changes likely related to an invariant delta found in this local static analysis are not definite or exhaustive. They are purely based on heuristics. But since they are only used to update change significance, the correctness of Algorithm 3 is not affected. After that, we update H^* to H^+ and recursively apply partition rules on H^*.

When H^+ fails T, either Fail-1 or Fail-2 applies, depending on the size of H^-. If $|H^-|$ is greater than one, as discussed before, the cause of test failure is not determined. We do not infer change significance in this case (Fail-2). Otherwise, we perform a similar analysis as in Pass and increase the significance scores of the related changes (Fail-1).

Termination Condition

The algorithm never attempts the same partition (H^+) twice and it terminates whenever H^* becomes empty or the 1-minimal condition defined in Definition 12 is met – $\forall \delta \in H^* \cdot (H^* \setminus \{\delta\}) \not\models T$.

5.3.2 Soundness and Completeness

The following theorem states that the algorithm is sound.

Theorem 2. *(Soundness). Given a history H and a test suite T, if the delta refinement algorithm terminates, then H^* is a 1-minimal semantics-preserving slice of H with respect to T.*

Proof. The soundness of the algorithm is straightforward. Since H^* is only updated when T is passed, H^* is always a valid semantics-preserving slice. The termination condition guarantees that it is also 1-minimal. □

As presented, the generic partition rules are non-deterministic. To ensure termination, we impose a notion of fairness on partition schemes. A *fair partition scheme* guarantees that a singleton partition for every atomic change in H^* is eventually reverted after very update of H^*. The following theorem states completeness of the algorithm.

Theorem 3. *(Completeness). Given a history H and a test suite T, the algorithm using fair partition schemes always terminates with finitely many rule applications.*

Proof. To see this, suppose the algorithm does not terminate. Since H^* has finite number of changes initially and its length is monotonically decreasing, $|H^*|$ has to eventually stay constant. Because of the fairness condition, every atomic change in H^* is eventually reverted and tested. If none of the tests pass, then the 1-minimal condition is met. If one of the tests passes, then $|H^*|$ should decrease. Both cases lead to contradictions. □

5.4 Implementation and Optimizations

In this section, we describe our implementation of a minimal semantic slicing tool based on the dynamic delta refinement algorithm and some of the optimizations applied.

5.4.1 Implementation

We implemented the delta refinement algorithm as a fully-automated precise semantic history slicing tool, Definer, for Java projects hosted in Git repositories.

We use JGit [JGi16], a Java implementation of Git, for repository manipulation and commit-level hunk dependency analysis (see Section 4.3.3). We use a modified version of ChangeDistiller [FWPG07] for extracting AST-level atomic changes from Git commits. We also use the Apache Byte Code Engineering Library (BCEL) [BCE15] to analyze dependencies among atomic changes, Daikon [EPG+07] for dynamic invariant detection, and Soot [VRCG+99] for performing local change impact analysis.

Definer is written in Java and yields fully-automated analysis of projects built with Maven [Mvn15]. The source code of Definer and all benchmarks used in our experiments are available online: `https://bitbucket.org/liyistc/gitslice`.

5.4.2 Optimizations

We now describe the optimizations and adaptations applied specific to Definer for dealing with more realistic cases.

Change Dependency Analysis

To avoid running into compilation errors, we perform a pre-analysis for each version in the history and compute direct dependencies for all changed AST nodes. This analysis produces a *multi-version change dependency graph* as shown in Figure 5.6.

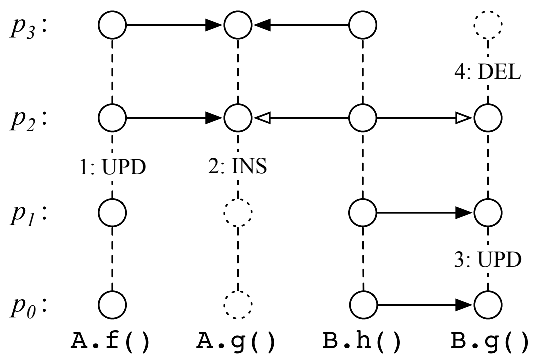


Figure 5.6: Illustration of change dependency analysis.

In this example, there are four program versions, i.e., p_0, p_1, p_2 and p_3, all of which are well-formed. There are three changed nodes, i.e., methods `A.f()` and `A.g()` which belong to class `A`, as well as `B.g()` which belongs to class `B`. There is also a fixed node `B.h()` which stays unchanged. Class `B` is a sub-class of `A`. Each node has a separate time-line on which its changes are labeled. In particular, `A.f()` has an update between p_1 and p_2; `A.g()` is inserted between p_1 and p_2; and `B.g()` is updated after p_0 but deleted after p_2. In Figure 5.6, solid arrows represent *necessary* dependencies while empty arrows represent *sufficient* dependencies. For instance, a method invocation of `g()` in `B.h()` makes `B.h()` necessarily depend on `B.g()` before p_2. But when `g()` is introduced in the super-class `A` in version p_2, both definitions of `g()` are sufficient dependencies of `B.h()`, i.e., existence of either one of them would satisfy the compilation requirement due to method inheritance.

This graph is useful for predicting compilation failures without actually compiling the program, as long as atomic changes for an AST node are reverted sequentially. For example, $\boxed{2}$ cannot be reverted from p_3 alone because both `A.f()` and `B.h()` necessarily depend on it. But $\{\boxed{1},\boxed{2},\boxed{4}\}$ can be reverted together since `A.f()` no longer depends on `A.g()` and the recovered `B.g()` substitutes the dependency for `B.h()`.

We build the multi-version dependency graph incrementally. A complete dependency graph is built by first analyzing the base version, and for subsequent versions it suffices to analyze only the changed classes.

Git Adaptation

The generic algorithm discussed in Section 5.3 operates on the level of atomic changes. To work with Git, we treat the set of atomic changes belonging to the same commit as a bundled group. The partition algorithm is adjusted such that changes in the same group always stay together and the significance score for a group is computed as the sum of the scores of its members.

Apart from dependencies between atomic changes, we also analyze dependencies between commits which are also known as the *hunk dependencies* (Chapter 4). Hunk dependencies for a commit are prerequisites which have to be in place so that the target commit can be applied without causing a Git merge conflict. Partitions of commits which do not comply with the hunk dependencies are discarded immediately without running any tests.

Other Optimizations

Ranking by Heuristics. Non-essential commits are commits which contain only *non-essential changes* [KR11], such as changes on test files, Javadocs and non-Java files including configuration files, change logs, etc. We assign negative initial significance scores for *non-essential commits* according to heuristics which gives higher priority for reverting them compared with other commits.

Lazy Dynamic Tracing. The software projects we experimented with are relatively large in size (30 to 190 KLOC). Instrumenting the entire project and tracing test execution end-to-end with Daikon is often impractical. But since we are only interested in the local direct impacts of changes, which give clearer signals for significance, we can simply trace classes that have changed during the input history.

5.5 Evaluation

In this section, we evaluate our implementation with respect to both its precision and performance using a benchmark suite obtained from real open-source software repositories. The goal of our empirical evaluation of Definer is to answer the following research questions:

RQ 3 How does the *precision* of history slices produced by Definer compare with those produced by CSlicer?

RQ 4 How *effective* is change significance ranking for guiding history partitions when compared with the basic partition scheme used by delta debugging?

RQ 5 How do different partition schemes affect the *performance* of Definer?

5.5.1 Subjects

Table 5.1: Statistics of the software projects used in the experimental evaluation of Definer.

Projects	#Files	LOC	#C-1y	#C-4m
`io`	227	29,173	127	42.3
`collections`	525	61,548	151	50.3
`math`	1,410	187,711	407	135.7

We tested Definer on a benchmark consisting of eight target functionalities selected from three open-source projects, namely Apache Commons IO Library (`io`) [IO16], Apache Commons Collections Library (`collections`) [Col16], and Apache Commons Mathematics Library (`math`) [Mat16]. These projects are all written in Java and their development histories are freely accessible online. They are also actively developed and maintained so that there are an abundance of new functionalities (e.g., features, bug fixes and improvements) to choose from. Statistics about each project is shown in Table 5.1. Columns "#Files" and "LOC" show the number of Java files and the total lines of code for each project, respectively. Column "#C-1y" shows the number of commits between January 1, 2015 and January 1, 2016 for each project. Column "#C-4m" shows the average number of commits over four months during the same period.

In order to test the history slicing capabilities of Definer, all of the experiments require an original history segment (H) and a target test suite (T) designated for certain high-level functionality. We randomly selected target functionalities and manually identified corresponding history segments based on commit messages and testing documents. In particular, we looked for commits which are accompanied by test suites intending to validate a functionality and selected such commits as the end points of the history analysis scopes.

According to our experience, the lifetime of a functionality typically spans around 100 commits which correspond to a period of one to four months for a project under active development. Therefore, the length of H was decided based on two factors:

- each program version in the chosen history should be well-formed and compilable as assumed by our change dependency analysis, and
- the average history length of a subject matches the average number of commits during a 4-month development period (∼76.1 commits).

Surprisingly enough, many versions in `math` and `collections` did not compile with the provided Maven configuration files. Even though non-compilable code does not affect the correctness of Definer, it could affect the soundness of the change dependency analysis and, thus, Definer's efficiency. Because of that, we test Definer under the well-formedness assumption and merge problematic commits with their children whenever possible to form a larger commit that lead to compilable versions.

Table 5.2: Descriptions of the experimental subjects used in the evaluation of Definer.

Projects	ID	Functionality Descriptions	End Point	$\lvert H\rvert$	$\lvert T\rvert$	$\lvert H^*\rvert$
`io`	I1	byte array output stream	89608628	150	3	28
	I2	identifies broken symlink files	b9d4976	50	1	3
	I3	file name utilities	63cbfe70	100	37	2
`collections`	C1	"index of" function in iterable utilities	90509ce8	100	1	12
	C2	"union" function in set utilities	9314193c	100	1	5
`math`	M1	error conditions in continuous output field model	6e4265d6	50	1	13
	M2	construct median with specific estimation type	afff37e0	50	1	2
	M3	large samples in polynomial fitter	b07ecae3	100	1	2

The details about each experimental subject are given in Table 5.2. Column "ID" lists the subject identifiers. Column "End Point" shows the target commits which correspond to the final version in our analysis scope. Columns "$|H|$" and "$|T|$" show the length of the original history segments and the sizes of the target test suites, respectively. Column "$|H^*|$" shows the length of the verified 1-minimal history slice.

5.5.2 Results

We conducted three experiments to address our research questions. The experiments were conducted on a desktop computer running Linux with an Intel i7 3.4GHz processor and 16GB of RAM. The results of the experiments are described below.

Experiment 5

Experiment 5 aims to compare Definer with CSlicer in terms of the precision of the produced history slices. We use the default configuration of Definer (Default) which adopts a simple partition scheme that reverts negative scored commits first whenever possible. The relative slice size for each subject is computed as the length of the produced history slice divided by the original history length, i.e., $|H^*|/|H|$. The results of the comparison are shown in Figure 5.7.

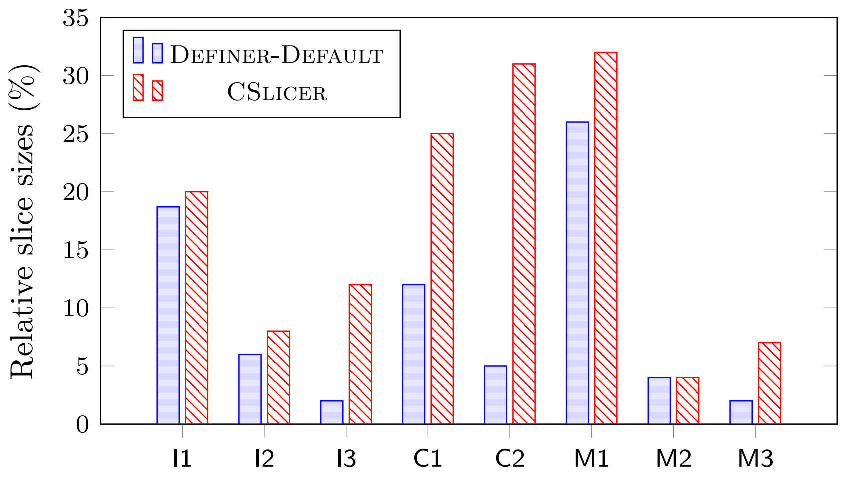


Figure 5.7: Sizes of slices: Definer vs. CSlicer in Experiment 5.

The history slices found by Definer are always shorter or equal to those computed by CSlicer (on average 45.6% shorter). In fact, all slices produced by Definer are verified to be 1-minimal while CSlicer does not guarantee minimality. For example, out of 100 commits from the subject C2, Definer finds a slice of length 5 which is much shorter than 31 reported by CSlicer. CSlicer took 51.6s to finish on average, while Definer took 1,161.7s. We consider this performance overhead to be reasonable since history slicing is often performed as an off-line maintenance task.

Experiment 6

Experiment 6 evaluates the effectiveness of using change significance ranking and change dependency analysis in speeding up the delta refinement loop. We compared three configurations of Definer in terms of the number of test runs needed to achieve 1-minimal solutions:

- Default as described before,
- Learn which only enables change significance learning and disables compilation failure prediction based on change dependency analysis, and

- Basic which also disables significance learning and thus is effectively equivalent to delta debugging which applies the basic (random) partition scheme.[1]

All configurations still apply hunk dependency analysis which predicts Git merge failures without actually picking the commits.

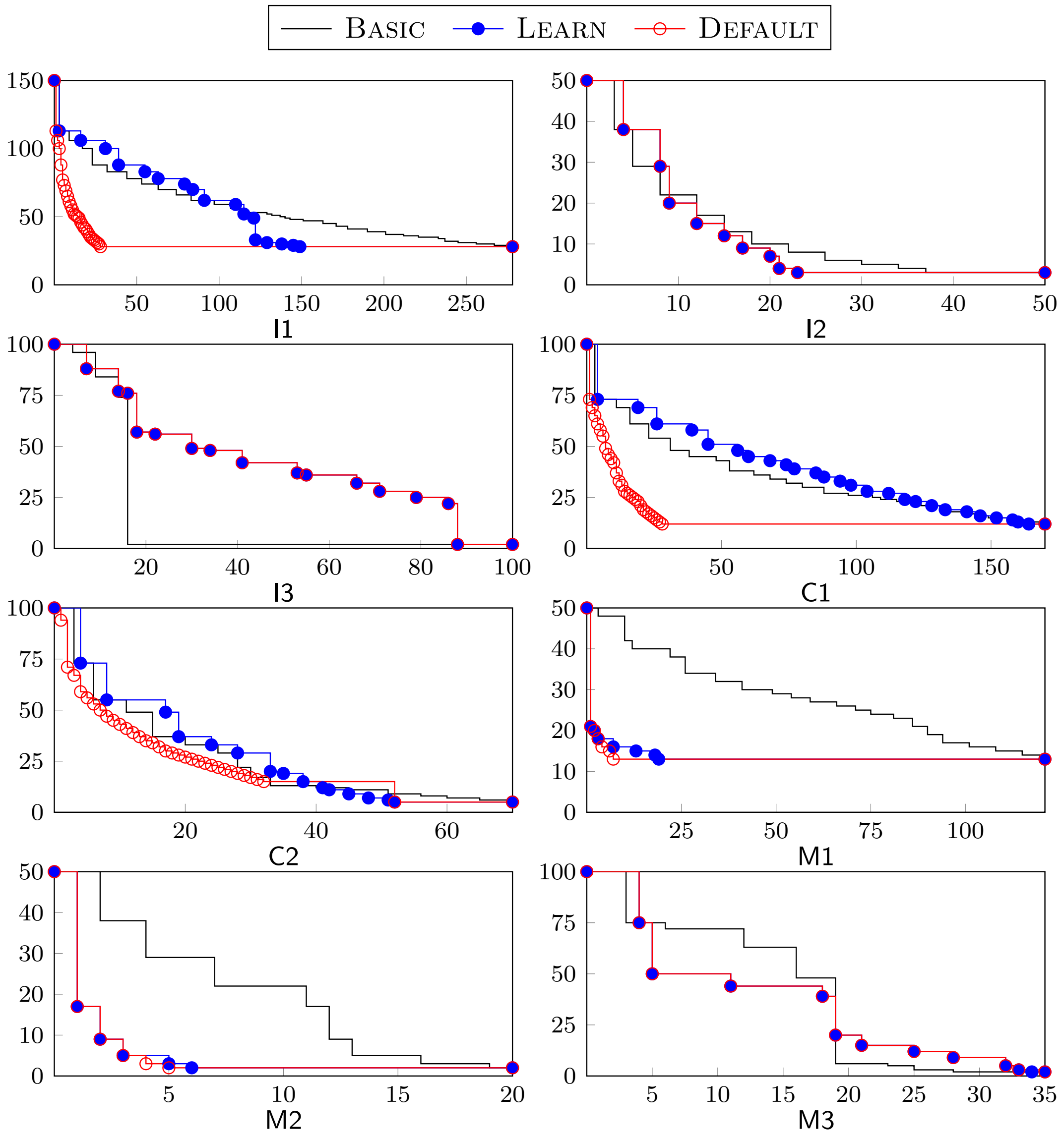


Figure 5.8: History reduction per test run in Experiment 6.

The results of the comparisons are shown in Figure 5.8 where the length of H^* (y-axis) is plotted as a function of the number of test runs (x-axis) used so far. In general, Default and Learn require fewer test runs than Basic to reach the minimal solution. In most of the cases, the advantage of significance learning is obvious, especially for cases such as I1, M1 and M2 where Learn requires on average only about 33% of test runs compared with Basic. In addition, change dependency analysis which prevents test from running on non-compilable programs helped extend this advantage further – it only takes about

[1]We could not directly compare with the original implementation in [Zel99] which does not work with Git repositories.

14% of test runs.

There is only a single case I3 for which the basic partition scheme performed much better: it was able to reduce a large chunk of commits quickly at the 17th test while the other two configurations did not reach minimal solution until the 88th test run. A closer look at I3 reveals that the minimal slice contains only 2 out of 100 commits, and most of the random partitions ended up being successful. In contrast, the Default partition scheme is more conservative and progresses more slowly in this case. Yet when the minimal slice is relatively large, for example in I1 (28 out of 150) and M1 (13 out of 50), the significance-guided partition is much more effective.

Experiment 7

We also experimented with three different partition schemes, namely Neg, NonPos, Low-3 and their combination, Combined. Recall that commits with positive significance scores are likely relevant to the target tests and vice versa; commits which do not have a score assigned cannot be classified until new signals become available.

All schemes follow the general steps described in Section 5.2.3 with different partition priorities at the beginning of each iteration. The Neg scheme only reverts commits which have negative scores. It is the most conservative one among the three. NonPos is the most aggressive one which reverts all commits with non-positive scores. Low-3 always reverts the lowest one third of the commits according to their significance ranking. Combined attempts all three partitions whenever possible. The results of Experiment 7 can be seen in Table 5.3, where each column lists the number of test runs required to reach the minimal solution and the best configurations for each row are in bold. All three partition schemes perform well on some of the subjects. For example, Low-3 required the smallest number of test runs for I1, I2 and C2. The combined scheme achieved the best overall performance by winning in 5 out of 8 examples.

Table 5.3: Comparisons of different partition schemes in Experiment 7.

ID	Neg	NonPos	Low-3	Combined
I1	258	258	**168**	**168**
I2	33	33	**25**	**25**
I3	89	**60**	89	89
C1	176	176	176	**172**
C2	72	72	**57**	**57**
M1	**27**	32	36	28
M2	**7**	8	11	**7**
M3	31	**23**	35	35

5.5.3 Evaluation Summary

To summarize, we evaluated the precision and performance of Definer empirically on a benchmark set of real-world software projects. We demonstrated that Definer produces more precise history slices than existing state-of-the-art techniques, such as CSlicer. Moreover, in the majority of cases, it outperforms the basic partition scheme used by delta debugging, thanks to the change significance ranking learned during the refinement process. With all optimizations combined, Definer achieves precise slicing results in an efficient manner.

5.6 Related Work

In this section, we compare our dynamic delta refinement algorithm with related work. We compare Definer with dynamic behavioral analysis and fault localization techniques.

5.6.1 Dynamic Behavioral Analysis

Through program instrumentation and execution tracing, dynamic analysis techniques [HL02, PE04, OX08] allow the comparison of precise runtime program behaviors. Daikon [EPG$^+$07] is one example of such techniques which discover likely program invariants from runtime executions. Daikon instruments the target program, traces variables of interest, and infers likely invariants for them. It has been widely used for many software developing tasks including debugging [BE04, DLE03], regression testing [XN05, PMH$^+$14], bug prevention [ECGN99] and more.

DIDUCE [HL02] is another tool for *dynamic invariant detection*. It trains a model for the target program by formulating hypotheses of invariants obeyed by the program and refining hypotheses dynamically through "presumably-good" runs. The produced model can be used to check for potential errors in other test runs. We use a similar idea of forming and updating hypotheses dynamically with multiple test executions. The key difference is that our goal is to infer change significance rather than program invariants. Therefore, we can exploit useful information from both passing and failing runs to improve the accuracy of our significance model.

Our work is also related to *behavioral regression testing* [OX08, JOX10] with respect to the usage of test executions for exposing behavioral differences across program versions. We differ in how the differences are used in the analysis: behavioral regression testing reports the results of behavioral comparison to users in order to help them complete and improve the quality of existing regression test suites [OX08, JOX10], whereas our goal is to speed up history slicing by identifying significant changes with the guidance from the behavioral differences.

5.6.2 Fault Localization

Delta debugging [Zel99] 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. However, in contrast to Definer which extracts semantic information from test results to guide its subsequent partition, delta debugging does not exploit this information and its partition scheme is fixed. Regarding slice quality, Zeller and Hildebrandt [ZH02] consider an approximated version of minimality, i.e., *1-minimal*, which guarantees that removing any single change set breaks the target properties. This trade-off on solution quality enables the authors to use an efficient divide-and-conquer search method.

Selective bisection debugging [SG17] is an enhancement over the traditional bisection debugging approaches based on binary search over software version history (e.g., Git-bisect [Git16a]). Bisection debugging is widely used in practice to identify bug introducing commits, but it can be expensive due to costly compilation and test execution process. Selective bisection debugging uses *test selection* and *commit selection* to reduce the number of tests to run and the number of commits to consider. In particular, commit selection uses test coverage information to predict if a certain commit in a bisection step is likely to be irrelevant to failing tests and skip the commit to save time. This is similar to Definer as they both rely on test signals to make predictions in the hope to reduce overall computational costs. Conceptually, selective bisection debugging tries to find which commit makes the test fail, while Definer tries to find which commits make the test pass. Another difference is that bisection debugging tries to locate a certain version while Definer attempts to find a 1-minimal history slice which is much harder ($\mathcal{O}(\log n)$ vs. $\mathcal{O}(n^2)$).

5.7 Summary

In this chapter, we proposed the dynamic delta refinement algorithm for finding minimal semantic history slices. We have implemented the algorithm as a prototype tool, Definer, which operates on Java projects hosted in Git. Definer largely improves the precision of the history slices over state-of-the-art techniques. The change significance learning techniques are also shown to be effective in speeding up the slicing process when applied to large scale software projects.

Chapter 6

Implementing a History Slicing Framework

"Practice is the sole criterion of truth."

— Fuming Hu

6.1 Introduction

Chapters 3–5 introduced the semantic history slicing problem and two different semantic slicing algorithms each having its pros and cons. In the process of experimenting with and evaluating our algorithms, we designed, developed, and refined a history slicing framework called CSlicerCloud. CSlicerCloud is a Web-based service tailored for Java projects hosted on GitHub. It has a user friendly front-end accessible from Web browsers and its server-side back-end unifies a collection of semantic history slicing techniques including CSlicer, Definer, and an approach combining both static and dynamic history slicing algorithms, called Combine.

In this chapter, we describe the architecture of CSlicerCloud, its user interface, prominent optimizations, and its evaluation on a dataset collected from open-source software projects. The main features of CSlicerCloud are:

- CSlicerCloud is a Web-based application built upon the Node.js JavaScript runtime [Nod17]. It requires no downloading, installation, or configuration. All parameters for performing history slicing can be configured through a graphical user interface.
- CSlicerCloud supports Java projects hosted on GitHub. For projects using Maven as their build system, the history slicing process is completely automated. It may require further configurations for other types of projects.
- The CSlicerCloud framework unifies both the static (Chapter 4) and the dynamic (Chapter 5) history slicing algorithms, which makes it precise and efficient at the same time.
- CSlicerCloud employs a number of important optimizations on both the slice quality and the slicing performance. The most prominent ones include hunk-level minimization, parallelization, and change significance indexing.

We evaluated CSlicerCloud and its many different configurations on a dataset [ZLRC17] of change histories collected from 10 open-source software projects. CSlicerCloud is accessible at: `http://www.cs.toronto.edu/~liyi/cslicer`. Its source code and configuration instructions are available at: `https://bitbucket.org/cslicer/cslicer-node`.

Organization

This chapter is organized as follows:

- In Section 6.2, we describe the architecture and implementation details of CSlicerCloud.
- In Section 6.3, we discuss the important optimizations implemented in the CSlicerCloud framework.
- In Section 6.4, we present the evaluation of the proposed optimizations.
- Finally, in Section 6.5, we conclude this chapter.

6.2 Architecture and Implementation

CSlicerCloud consists of a client-side front-end and a server-side back-end. The front-end is accessible from Web browsers and is implemented on top of the Node.js JavaScript runtime [Nod17]. The back-end handles user requests from the front-end, performs the actual history slicing jobs, and sends back responses with the slicing results. See Figure 6.1 for an architectural overview.

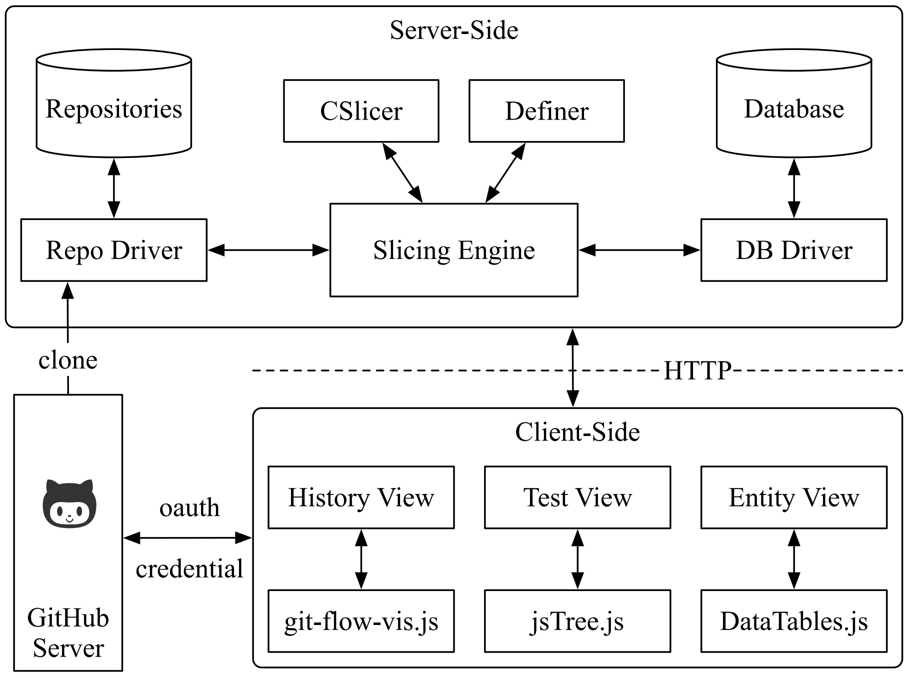


Figure 6.1: Architecture of CSlicerCloud.

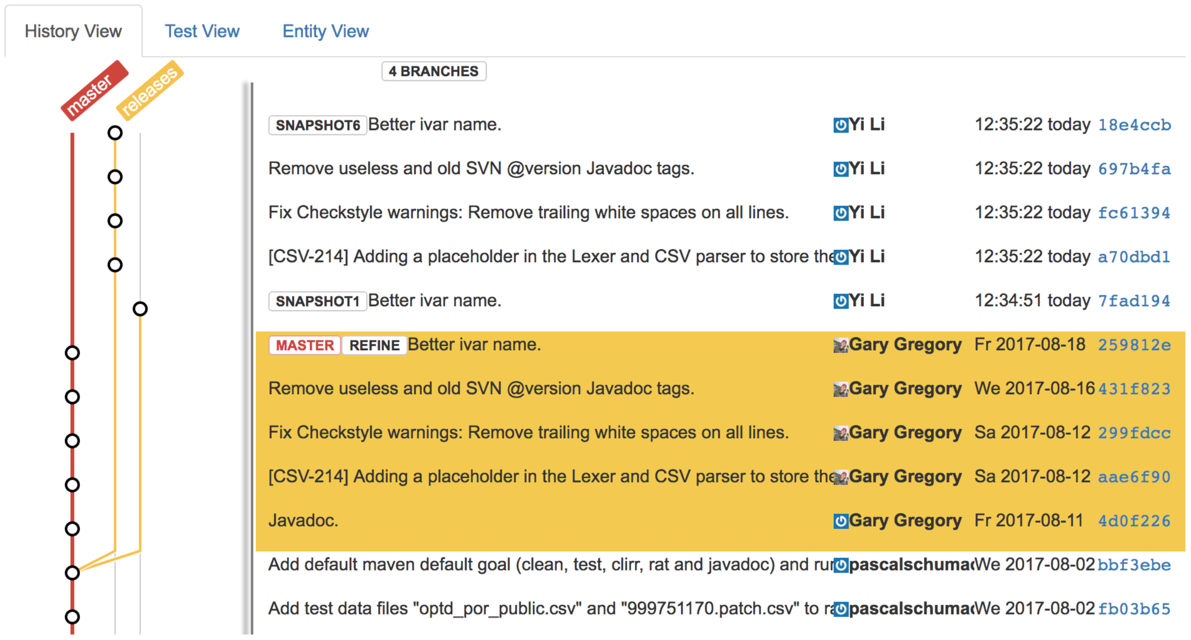
From the high level, the client-side running on browsers obtains user credentials from GitHub [Git17b] server through the OAuth protocol [Har12]. Then the GitHub credentials are used to identify users and access their repository meta data. The client-side also provides graphical user interfaces and handles user interactions. The requests from users are then sent to the server-side through HTTP connections. The server-side retrieves repository data from the GitHub server, and the slicing engine performs history slicing as per the users' requests. The slicing results are then communicated back to the client-side with HTTP messages. The server-side also performs indexing of version histories and caching of slicing results, storing them in the database. Sections 6.2.1 and 6.2.2 discuss the implementation details of the front-end and the back-end, respectively.

6.2.1 Front-End Implementation

The front-end interacts with users through three different views – namely, the *history view*, *test view* and *entity view*. We now describe each view in detail.

History View

The history view visualizes the version histories of a given software repository and allows users to define a history range for the subsequent history slicing tasks. Figure 6.2 is a screen shot of the history view displaying the version histories of the Apache Common CSV project [CSV17]. The left pane depicts the branches and commits graphically, while the right pane lists other meta data including the log messages, authors, and the SHA-1 commit IDs of the corresponding commits. Each item in the commit list is clickable such that a range can be defined by selecting a start commit and an end commit. The history view also displays slicing results when the history slicing process is finished. The commits of a computed history slice are highlighted as feedback to the users (see Figure 6.2).



SNAPSHOT6 Better ivar name.	Yi Li	12:35:22 today	18e4ccb
Remove useless and old SVN @version Javadoc tags.	Yi Li	12:35:22 today	697b4fa
Fix Checkstyle warnings: Remove trailing white spaces on all lines.	Yi Li	12:35:22 today	fc61394
[CSV-214] Adding a placeholder in the Lexer and CSV parser to store the	Yi Li	12:35:22 today	a70dbd1
SNAPSHOT1 Better ivar name.	Yi Li	12:34:51 today	7fad194
MASTER REFINE Better ivar name.	Gary Gregory	Fr 2017-08-18	259812e
Remove useless and old SVN @version Javadoc tags.	Gary Gregory	We 2017-08-16	431f823
Fix Checkstyle warnings: Remove trailing white spaces on all lines.	Gary Gregory	Sa 2017-08-12	299fdcc
[CSV-214] Adding a placeholder in the Lexer and CSV parser to store the	Gary Gregory	Sa 2017-08-12	aae6f90
Javadoc.	Gary Gregory	Fr 2017-08-11	4d0f226
Add default maven default goal (clean, test, clirr, rat and javadoc) and ru	pascalschuma	We 2017-08-02	bbf3ebe
Add test data files "optd_por_public.csv" and "999751170.patch.csv" to r	pascalschuma	We 2017-08-02	fb03b65

Figure 6.2: The history view of CSlicerCloud. Slicing results are highlighted.

The graphical visualization of Git version histories is powered by the JavaScript library GitFlowVisualize [Git17a]. GitFlowVisualize accepts commit and branch data as JSON [JSO17] objects, which are parsed from the target project Git repositories. We have modified the library to enable history selection and results highlighting.

Test View

The test view visualizes the unit tests of a Java projects, and users can select the tests they want to use as criteria for the history slicing tasks. Figure 6.3 is a screen shot of the test view showing the test cases organized as an expandable tree structure. Each top-level node represents a Java test file, for example, "`CSVFileParserTest.java`" (Figure 6.3). The leaf nodes visible when a test file node is expanded represent the test methods contained in the test file. For example, the "`CSVFileParserTest.java`" has two test methods – namely, "`testCSVFile`" and "`testCSVUrl`" (Figure 6.3). Each method node also comes with a check box which allows it to be selected as one of the slicing criteria.

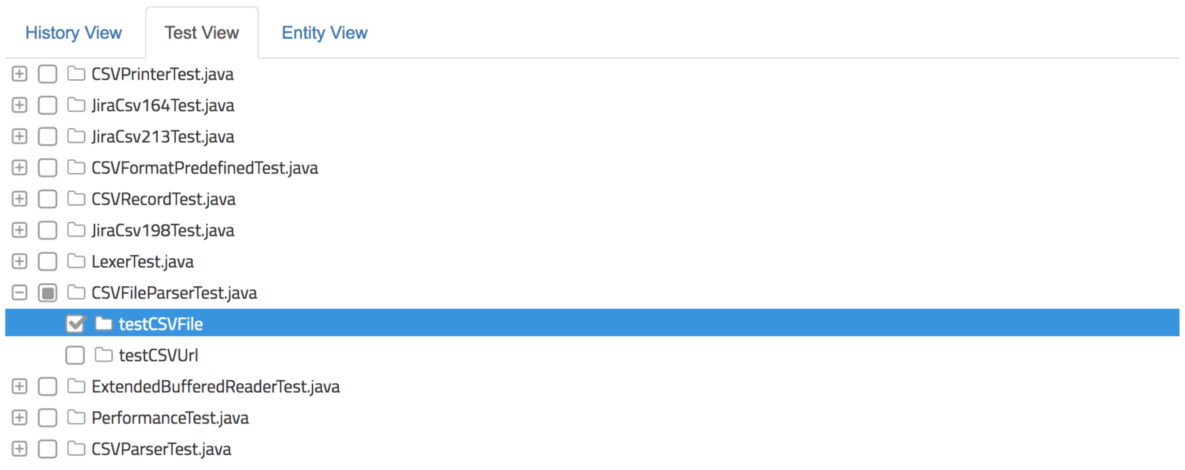


Figure 6.3: The test view of CSlicerCloud.

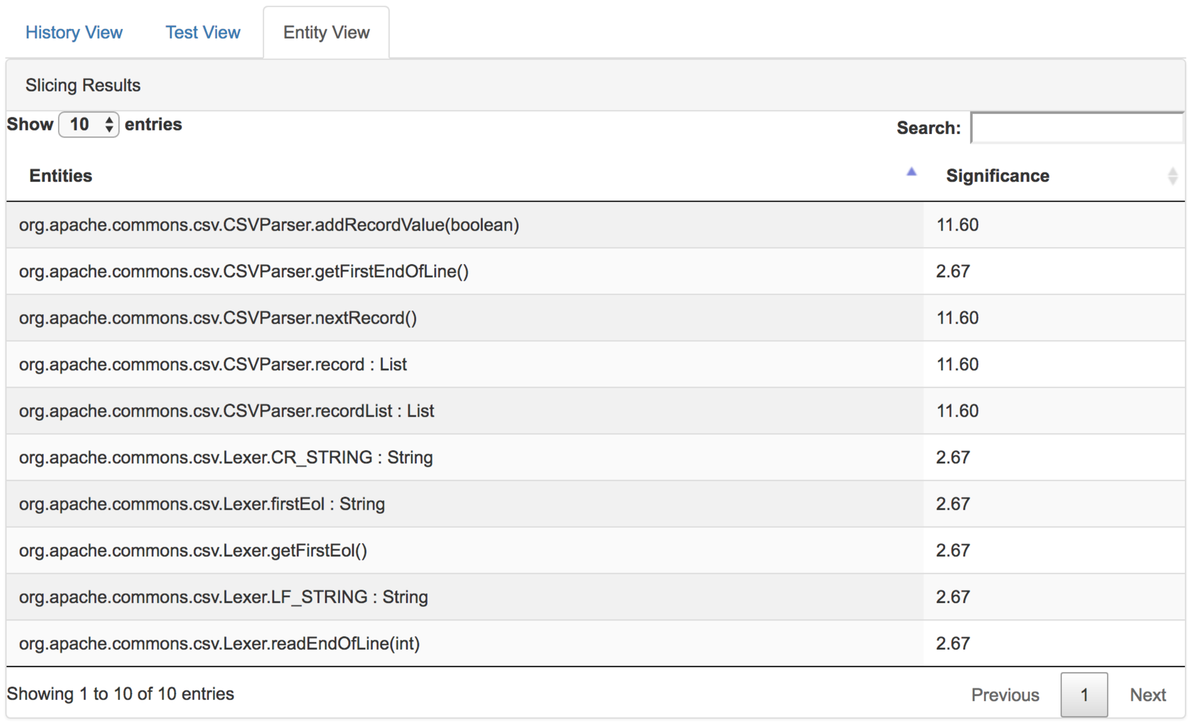
The test view is built on top of another library, jsTree [JST17], which provides an interactive tree in JavaScript. The data sources are provided by the server-side as JSON objects. We will explain how the test cases are automatically extracted from project repositories in Section 6.2.2.

Entity View

The entity view displays the code entity-level significance scores (see Section 5.3) inferred from running the dynamic history slicing algorithm using "`CSVFileParserTest.testCSVFile`" as the slicing criterion. The significance score indicates the estimated relative importance of a code entity with respect to the target tests for a particular input history.

Figure 6.4 shows an example entity view which displays the significance scores in a table. For instance, the first row in the table shows that the method "`CSVParser.getFirstEndOfLine()`" has a significance score of about 2.67. Since the score represents the relative importance of an entity compared with the others, the absolute value here carries little meaning except that a positive score indicates that the changes on the entity can potentially affect the test results. The other five code entities modified within the analyzed history range have the same significance score.

The entity view is implemented based on a JavaScript data visualization library DataTables [Dat17]. It supports flexible viewing options such as sorting, searching, and multi-page views. The outputs from



Entities	Significance
org.apache.commons.csv.CSVParser.addRecordValue(boolean)	11.60
org.apache.commons.csv.CSVParser.getFirstEndOfLine()	2.67
org.apache.commons.csv.CSVParser.nextRecord()	11.60
org.apache.commons.csv.CSVParser.record : List	11.60
org.apache.commons.csv.CSVParser.recordList : List	11.60
org.apache.commons.csv.Lexer.CR_STRING : String	2.67
org.apache.commons.csv.Lexer.firstEol : String	2.67
org.apache.commons.csv.Lexer.getFirstEol()	2.67
org.apache.commons.csv.Lexer.LF_STRING : String	2.67
org.apache.commons.csv.Lexer.readEndOfLine(int)	2.67

Showing 1 to 10 of 10 entries

Figure 6.4: The entity view of CSlicerCloud.

the slicing engine are parsed as JSON objects and then used as the data source.

6.2.2 Back-End Implementation

The server-side back-end consists of three components: the *slicing engine* which accepts history slicing tasks and dispatches them to the appropriate history slicing techniques, namely CSlicer and Definer. The slicing engine also delegates the repository related jobs such as retrieving and processing project version histories, compiling source code, and running tests to the *Repo driver*. On the other hand, the *DB driver* acts as a middleware between the slicing engine and the database which provides a persistent storage for previously computed results.

Slicing Engine

The slicing engine is the core of the server-side. It acts as a wrapper of the underlying history slicing techniques, not to simply dispatch slicing job requests coming from multiple users, but to choose the best strategy of combining the slicing algorithms for the best precision and efficiency. Based on the size of the job request, influenced by factors such as the length of the input history and the complexity of the target tests, the slicing engine decides whether to run CSlicer and Definer exclusively, or to run them in a combined way as will be discussed in Section 6.3.2.

When in idle, the slicing engine indexes cached version histories in order to speed up future history slicing computation. The significance score of changes with respect to a test case can also be precomputed and stored for future use. Details on history indexing and results caching will be discussed in Section 6.3.4.

Repo Driver

The Repo driver hides the low-level details of repository-related operations and provides an abstracted interface for the slicing engine. The supported high-level operations include cloning repositories from the GitHub server, duplicating repositories for parallel test execution (see Section 6.3.3), compiling source code with Maven, building test case lists, running tests with Surefire [Sur17] and processing test results. The Repo driver uses an asynchronous model which does not block the progress of the clients and allows callbacks to be registered for time consuming operations.

In order to extract the list of available test cases and display it in the test view, we first check out the repository to the selected "end commit". We then compile the tests into byte code and use the BCEL [BCE15] library to analyze the byte code and output the fully qualified test method names as JSON objects.

DB Driver

Similarly, the DB driver supports writing to and querying from the database. There are two tables maintained in the database – namely, the "*runs*" and "*significance*" tables. The "runs" table records the parameters as well as results for each history slicing run, and has the following fields:

- `repo_path`: the local path to the repository on the server-side,
- `start_commit`: the SHA-1 ID of the first commit of the chosen history range,
- `end_commit`: the SHA-1 ID of the last commit of the chosen history range,
- `tests`: the fully qualified names of the chosen target tests (we assume these are unique),
- `engine`: the strategy used for performing the slicing task,
- `result`: the slicing result.

The "significance" table records the significance scores of all analyzed code entities and has three fields:

- `test`: the fully qualified name of the target test,
- `entity`: the name of the changed entity,
- `score`: the accumulated significance score of the changed entity with respect to the target test.

If a subsequent slicing request matches with one of the entries in the "runs" table, the slicing result is directly returned as cached in the database. Otherwise, the significance scores of the relevant entities are queried from the "significance" table to initialize a new slicing run. The value stored in the `score` field is an estimation of the importance of a code entity based on all previous slicing runs.

6.3 Prominent Optimizations

In this section, we present the optimizations which we applied for improving both the precision and the performance of the history slicing algorithms. The optimizations on slicing quality include bringing the history slicing algorithms to the hunk level which yields much better precision and applying slice minimization which guarantees true minimality of the produced history slices. The optimizations on slicing performance include the parallelization of test runs and the indexing of change histories.

6.3.1 Hunk-level History Slicing

Using commit as the smallest unit for doing history slicing has the benefit of preserving the traceability from the high level semantic property to its corresponding commit-level meta information such as authors, change dates, and log messages. This information can be useful in supporting other downstream maintenance tasks.

In practice, commits usually contain changes to many files and multiple classes and methods, organized as hunks. A *hunk* is the smallest unit of code change in language-agnostic version control systems [FAW09]. Different hunks in the same commit are not necessarily logically related or relevant to the same feature. Thus, considering a commit as an atomic unit does not allow us to remove many unnecessary changes for the target features.

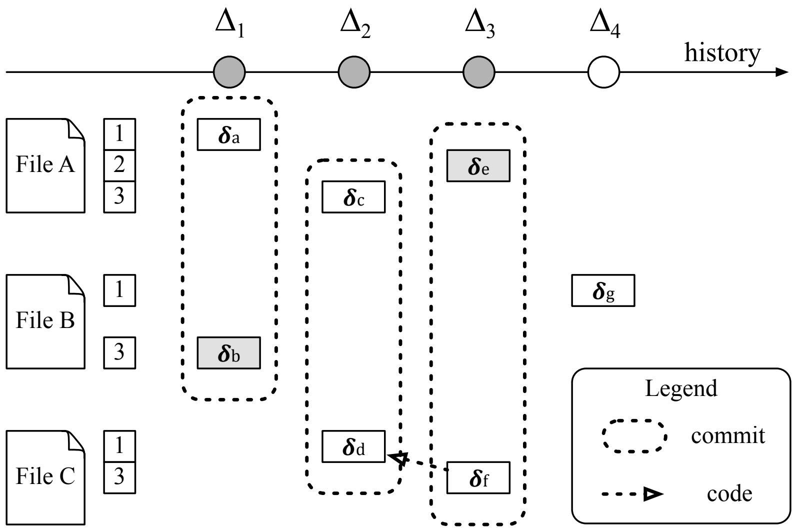


Figure 6.5: An illustration of the sources of imprecision in commit-level history slicing.

Example 5

Figure 6.5 shows a diagram illustrating the sources of imprecision in commit-level history slicing. The history segment H contains four commits, i.e., $H = \langle \Delta_1, \Delta_2, \Delta_3, \Delta_4 \rangle$. Each commit can be further broken into a set of hunks potentially spanning over multiple files. For instance, Δ_1 has two hunks, δ_a and δ_b, over files A and B, respectively.

The only feature-related changes in this example are δ_b and δ_e, shaded in gray. However, when performing history slicing on a commit level, we have to inevitably include unnecessary changes due to *commit dependencies* – two hunks in the same commit are *commit-dependent* on each other. For example, δ_a is included because of δ_b, and δ_f is included because of δ_e (commit bundles are depicted as dashed boxes in Figure 6.5).

Unnecessary changes introduced by commit dependencies can induce further imprecisions. For example, δ_f relies on an earlier hunk δ_d in order to function correctly (shown as a dashed arrow in Figure 6.5). This is known as a *code dependency* – including dependencies between a child and parent code entities, between a variable usage and definition, etc. The inclusion of δ_f therefore forces us to include δ_d as well, due to code dependency.

Thus, with commit-level history slicing, the best result achievable is a sub-history of length three: $\langle \Delta_1, \Delta_2, \Delta_3 \rangle$. In contrast, hunk-level history slicing allows us to reduce the number of unnecessary changes, resulting in δ_b and δ_e, as intended.

Implementing Hunk-level History Slicing

To implement hunk-level history slicing, the only change we need to make to the original history slicing algorithms (Algorithms 1 and 3) is to add an additional preprocessing step which splits commits into hunks. It is necessary to keep the mapping from the original commits to the corresponding set of hunks so that the traceability of historical meta-data can be restored.

Algorithm 4 An algorithm for splitting commits into hunks.

```
Require: $H \neq \langle\rangle$
Ensure: $H(p_0) = H_{hunks}(p_0)$
 1: procedure SplitCommits($p_0, H$)
 2:     $m, H_{hunks} \leftarrow \emptyset, \langle\rangle$
 3:     for $\Delta_i$ in $H$ do
 4:         $m(\Delta_i) \leftarrow$ SplitOneCommit($p_{i-1}, p_i, \Delta_i$)
 5:         $H_{hunks} \leftarrow H_{hunks}, m(\Delta_i)$
 6:     end for
 7:     return $(m, H_{hunks})$
 8: end procedure

Require: $\Delta(p) = p'$
Ensure: $hunks(p) = p'$
 9: procedure SplitOneCommit($p, p', \Delta$)
10:     $hunks \leftarrow \langle\rangle$
11:     Checkout program version $p'$                                  ▷ git checkout $p'$
12:     Unstage all hunks of commit $\Delta$                              ▷ git reset $p$
13:     while there is unstaged change do
14:         Stage a single hunk                                          ▷ git add -p
15:         Commit the hunk as $\delta_i$, with a log message indicating its origin   ▷ git commit
16:         $hunks \leftarrow hunks, \delta_i$
17:     end while
18:     return $hunks$
19: end procedure
```

Algorithm 4 outlines the process of splitting a change history H into an equivalent history H_{hunks} which contains only hunks (see the procedure SplitCommits). A mapping m from commits in H to hunks in H_{hunks} is also returned. The procedure iterates through all the input commits in sequence (Line 3 to 6) and calls a sub-routine SplitOneCommit for the actual splitting.

The procedure SplitOneCommit splits a single commit into a sequence of commits each containing only a single hunk. The effects of applying the hunks in sequence is equivalent to applying the original commit. To process the target commit Δ, we first check out the version p' right after its application

(Line 11). Then we use "`git reset`" to unstage all hunks of the commit in order to make them separate commits afterwards (Line 12). The Git command "`git add -p`" provides a way of staging individual hunks (Line 14) and they can then be committed individually. Finally, the sequence of hunks is returned (Line 18).

6.3.2 Minimizing Slicing Results

As discussed in Section 3, enumerating all sub-histories of a history H is highly unrealistic when the length of H is large (requiring in worst case $2^{|H|} - 1$ test runs). In contrast, a minimal slice of H can be derived more efficiently from a much shorter sub-history, e.g., a semantic slice H' returned by Algorithm 1 (now requiring in worst case $2^{|H'|} - 1$ test runs). However, the cost of a direct enumeration of H' can still be prohibitive. We propose two heuristic-based techniques that can be applied in combination to reduce false positives in H' and minimize H' much more efficiently: (S1) *change pattern matching* which analyzes changes statically and filters out common false positives according to predefined patterns; and (S2) *sub-history enumeration* which takes hunk dependencies into consideration and efficiently examines only cherry-pickable sub-histories, i.e., sequence of commits that can be applied without causing merge conflicts.

The overall minimization workflow is given as the procedure Minimize in Algorithm 5. Minimize takes as input a base version program p_0, a semantics-preserving history slice H and the target test suite T. Since the static pattern matching approach is much cheaper than enumerating all sub-histories, we first use MatchPattern as a pre-processing step to opportunistically shorten H by filtering out commits containing only "insignificant" changes (Lines 3-4). Then the intermediate result H_{S1} is verified against the tests. If tests pass, then the procedure Enumerate is called to enumerate and verify all cherry-pickable sub-histories of H_{S1} (Line 6). Otherwise, we enumerate the original input history H instead (Line 7). The enumeration procedure can be terminated prematurely based on the available resources and still return a valid slice with best-effort.

Static Pattern Matching (S1)

The AST differencing algorithm used in Algorithm 1 (see Chapter 4) treats each method as a single structural node. To detect local refactorings and unobservable changes within method bodies, we apply a finer-grained differencing algorithm at the statement-level granularity and then categorize atomic changes according to their *significance level* [FG06]. Similar to the definition in Fluri and Gall [FG06], we consider an atomic change as less significant if the likelihood of it affecting the test results or other code entities is low. We opportunistically drop low-significance changes if they happen to match predefined patterns.

Some examples of the patterns include local refacotring/rewriting, low impact modifier changes such as removal of the `final` keyword and update from `protected` to `public`, as well as white list statement updates such as modifications to printing and logging method invocations. We also allow users with domain knowledge to provide insights on which components (such as classes, methods, fields and statements) do not affect the test results to further prune the functional sets. These patterns are generally applicable to different code bases. More project-specific rules and white lists can also be devised with insights from the project developers. The low significance changes are processed separately, and we provide users with the options to keep or exclude them as they wish.

Algorithm 5 An algorithm for finding minimal semantic slice of a given history H.

```
Require: $H(p_0) \models T$
Ensure: $\forall H_{sub} \lhd H^* \cdot (|H_{sub}| < |H^*|) \implies \neg(H_{sub} \lhd_T H)$
 1: procedure Minimize($p_0, H, T$)
 2:     $H_{S1} \leftarrow H$
 3:     for all $\Delta \in H_{S1}$ do
 4:         if MatchPattern($\Delta$) then $H_{S1} \leftarrow H_{S1}/\{\Delta\}$
 5:     if $H_{S1}(p_0) \models T$ then
 6:         $H^* \leftarrow$ Enumerate($p_0, H_{S1}, T$)
 7:     else $H^* \leftarrow$ Enumerate($p_0, H, T$)
 8:     return $H^*$
 9: end procedure

10: procedure Enumerate($p_0, H_{S2}, T$)
11:     $L \leftarrow Sort$(Pickable(HunkGraph($H_{S2}$)))
12:     for all $H_{sub} \in L$ do
13:         if $H_{sub}(p_0) \models T$ then return $H_{sub}$
14:     return $H_{S2}$                                  ▷ no shorter history found
15: end procedure

Require: $(V, E)$ is a DAG
Ensure: $\forall H_{sub} \in L \cdot H_{sub}$ is cherry-pickable on $p_0$
16: procedure Pickable($V, E$)
17:     if $|V| = 1$ then return $\{\langle\rangle\}$
18:     $L \leftarrow \emptyset$
19:     $R \leftarrow$ RootNodes($V, E$)
20:     for $r \in R$ do
21:         $V \leftarrow V/\{r\}$                          ▷ remove a root
22:         $E \leftarrow E/Out(r)$                         ▷ remove out edges
23:         $L \leftarrow L \cup (\{\overline{V}\} \cup$ Pickable($V, E$))
24:     end for
25:     return $L$
26: end procedure
```

Dynamic Sub-history Enumeration (S2)

Several types of oracle-unobservable changes cannot be matched using predefined patterns. More sophisticated static analysis such as program slicing [Tip95] is able to identify more precisely the set of program statements (a program slice) which have the potential to affect the test oracle. However, in practice, a program slice does not always subsume an oracle-observable set, namely, a change outside of the program slice can still affect the test results. This is because traditional program slicing often fails to detect dependencies outside the control of the program itself, such as dependencies due to configuration files [BGH+15].

The most precise and reliable approach for minimizing history slices is through the exhaustive enumeration and verifying the test results directly when the number of candidates is small. When hunk dependencies are present, not all sub-histories are cherry-pickable on text-based SCM systems. Instead of enumerating all sub-histories of H, which is exponential to the length of H, we only consider cherry-pickable sub-histories that can be verified through test runs. This insight enables us to find minimal solutions for many examples in our benchmark (cf. Section 6.4.1). The procedure Pickable in

Algorithm 5 uses an approach similar to the Kahn's topological sorting algorithm [Kah62] to generate all cherry-pickable proper sub-histories for a given history slice.

The idea behind the procedure Pickable (Lines 16-26) is as follows. The hunk dependencies among commits can be represented using a directed acyclic graph (DAG) where V is the set of vertices (commits) and E is the set of edges (hunk dependencies). There is an edge pointed from v_1 to v_2 if and only if v_1 directly or indirectly hunk-depends on v_2. A commit is not cherry-pickable if any of its hunk dependencies is missing. The algorithm prevents this from happening by starting from V and only removing vertices that have no incoming edge (Line 19). After a vertex is removed, the remaining history is added to the current set of cherry-pickable sub-histories L, and the remaining graph is processed recursively.

Lemma 3. *(Soundness of* Pickable*).* Pickable(V, E) *returns all cherry-pickable proper sub-histories of* H_{S2}.

Proof. The proof is by induction. Considering the base case where $|V| = 1$, the only proper sub-history is $\langle\rangle$. Now suppose Pickable(V, E) returns all cherry-pickable proper sub-histories for $|V| \leq k$. At any stage, only root nodes (vertices with no outgoing edges) can be removed without breaking the hunk dependencies. When $|V_{k+1}| = k+1$, removing either one of the root nodes produces a cherry-pickable proper sub-history, $\overline{V_{k+1}/r}$, where the overhead bar stands for a sequence of change set derived from the vertices. Taking the union of each case gives us $\bigcup_{r \in \textsc{RootNodes}(V_{k+1}, E_{k+1})} \{\overline{V_{k+1}/r}\} \cup \textsc{Pickable}(V_{k+1}/r, E_{k+1}/Out(r))$. □

Theorem 4. *(Correctness of Algorithm 5).* Minimize(p_0, H, T) *returns a minimal semantic slice* H^* *such that* $\forall H_{sub} \lhd H^* \cdot (|H_{sub}| < |H^*|) \implies \neg(H_{sub} \lhd_T H)$.

Proof. Assume only cherry-pickable sub-histories are considered.[1] The theorem trivially holds given Lemma 3 and the fact that the sub-histories are traversed in increasing order of their lengths. □

Algorithm 5 always terminates since there are only finitely many sub-histories. However, in practice, enumerating all combinations can still be time consuming even with the preprocessing step. We report our empirical findings in Section 6.4.1.

6.3.3 Parallelizing Test Runs

Apart from improving the quality and precision of the slicing results, we also applied several optimizations to scale the computation of semantic history slices. Algorithm 3 presented in Chapter 5 gives a general framework of the dynamic delta refinement characterized as a set of generic rules. The execution of the algorithm alternates between two major stages:

- a *partition stage* which produces history slices to be tested for the preservation of the desired semantic properties, and
- an *execution and learning stage* which runs tests on the partitioned history slices and infers change significance.

The key insight used in this optimization is that the majority of the running time of Algorithm 3 is spent repeatedly executing tests and waiting for the test results. By parallelizing test runs, we can significantly speed up the overall slicing computation given more computing resources. In fact, Algorithm 3

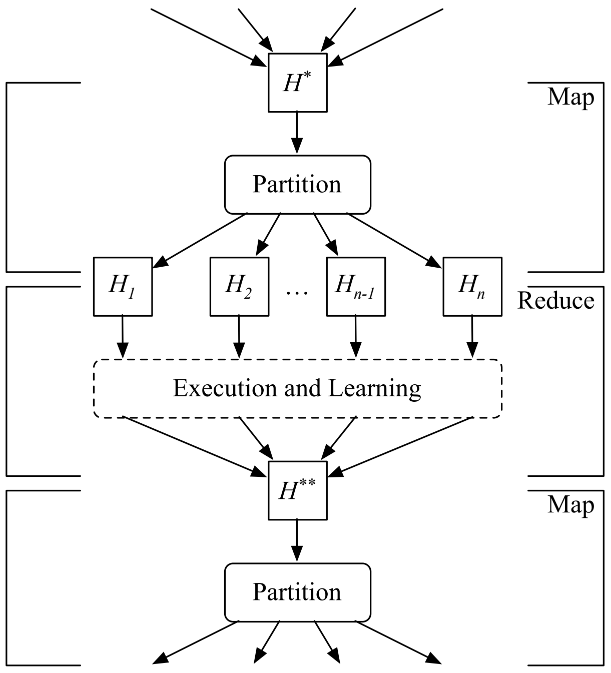


Figure 6.6: Illustration of the parallelization of test runs.

does exhibit inherent parallelism in its structure, which can be exploited in a MapReduce-like [DG04] fashion.

Figure 6.6 illustrates the high-level work flow of Algorithm 3 with parallelization enabled. During the map stage, the current minimal semantics-preserving slice (H^*) is partitioned into several candidate history slices ($H_1, H_2, \ldots, H_{n-1}, H_n$). The produced candidates are then processed by multiple execution and learning threads simultaneously in the reduce stage. We make n copies of the repository to avoid conflicts between threads. Depending on the outcome of the tests, we either redo partition of H^* (with a finer granularity) if all tests fail (not shown in Figure 6.6), or update the current minimal semantic slice to the candidate passing the test (H^{**}). The algorithm keeps iterating and performing the map and reduce stages until a 1-minimal semantic slice is found. Empirical evaluation of the optimization is presented in Section 6.4.

6.3.4 Indexing and Caching

CSlicerCloud uses its offline time to index version histories in which the user is most likely to be interested in the future. The precomputed change significance scores are indications of the correlation between individual code entities and test results; the scores can be used to guide the initial partition of subsequent runs of Algorithm 3.

Figure 6.7 depicts the history indexing work flow. The dashed arrows represent taking a specific item from the collection. First, as mentioned in Section 6.2.2, the result of each history slicing run is cached in the database. When the slicing engine is idle, it takes a single test t and an individual atomic change δ, and checks if the removal of δ affects the result of t. If t passes without δ, the database is updated with the decreased significance score of the changed code entity, and vice versa. Previous user requests

[1]This assumption can be relaxed in semantic-based SCM systems.

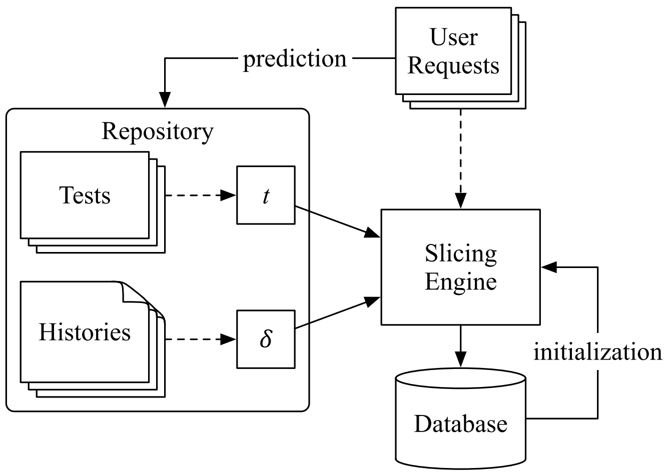


Figure 6.7: Illustration of the history indexing and results caching.

are used for predicting future ones – tests previously chosen by users as slicing criteria and changes of close proximity with previously chosen commits are given high priority when indexing.

6.4 Evaluation

In this evaluation of our optimization techniques, we aim to answer the following research questions:

RQ 6 How effective are the *slice minimization techniques* in CSlicerCloud?

RQ 7 How does CSlicerCloud perform comparing with delta debugging?

In order to better understand the benefits and limitations of the slice minimization techniques, we used a version of CSlicerCloud which has two phases, with Phase 2 performing minimization on top of the results from Phase 1:

Phase 1 CSlicer without optimizations introduced in Chapter 4; and

Phase 2 slice minimization introduced in Section 6.3.2.

6.4.1 Experiment 8: Slice Minimization

Experiment 8 is designed to answer RQ 6 by running CSlicerCloud in the minimization mode. We are interested in the number of false positives that can be reduced by both the static pattern matching and the dynamic sub-history enumeration techniques.

Subjects and Methodology. In order to study the effectiveness of our slice minimization techniques, we chose a subset (10 out of 20) of the benchmarks used in Experiment 4 (see Chapter 4), which have relatively short semantic slices (the number of FUNC and COMP commits is smaller than or equal to 11) so that we could exhaustively enumerate all sub-histories to establish the ground truth. For each of them, we exhaustively enumerated all cherry-pickable sub-histories and found at least one minimal semantic slice. The length of slices before and after minimization is given in Table 6.1.

For each of the subjects, we first applied static pattern matching (S1) to identify and eliminate change sets with low significance. This includes two local code refactorings, seven white list statement updates, and four low significance changes. Then we applied dynamic sub-history enumeration (S2) on the remaining history slices to find a minimal solution. Note that in our experiments, all minimal solutions were found after the S1 step. In a more general case, no solution might exist after this stage because S1 could eliminate an essential commit. Then we need to exhaustively enumerate all sub-histories of the original slice before the S1 reduction (see Algorithm 5 for details).

Table 6.1: Sizes of history slices produced at various stages in Experiment 8.

Subject	$\|H'\|$	$\|H^*\|$	S1	S2	T1	T2
H1	3	2	1	0	2	2
H2	2	2	0	0	3	3
A2	4	2	1	1	7	5
E2	9	1	6	2	5	2
M3	3	3	0	0	6	6
M4	4	2	0	2	3	3
I2	2	1	0	1	3	3
I3	6	2	1	3	12	8
L1	2	2	0	0	3	3
L2	6	2	0	4	8	8

Results. The detailed experimental results are reported in Table 6.1. Columns "$|H'|$" and "$|H^*|$" show the lengths of slices produced in Experiment 4 and the minimal slices, respectively. Columns "S1" and "S2" list the number of reduced commits by each technique. Columns "T1" and "T2" list the number of test runs required in order to find the minimal sub-history with and without using S1, respectively. For instance, the original slice for E2 has nine commits. Applying S1 allows us to remove six commits and to minimize the remaining three commits using S2 we enumerate all singletons, then all pairs, and so on. The actual number of test runs used for this case is two: one failed test on the empty history slice, and one successful test on the first try of the singleton slices. In contrast, without applying S1, the minimization needed five test runs.

As a result of our experiment, we were able to prove that the slices produced for H2, M3 and L1 are already minimal by enumerating all their candidate sub-histories with three, six and three failed executions, respectively. For the rest, we found minimal solutions on the first few tries.

Also, comparing Columns S1 and S2, we conclude that the heuristic-based change filtering patterns are both effective and generally applicable to different subjects in reducing false positive commits. Notably, six commits were filtered out during the S1 step for E2. In addition, applying S1 significantly reduced the number of sub-histories which need to be enumerated for S2.

Finally, taking into account hunk dependencies existing inherently among commits in the input slices helps mitigate the exponential explosion in sub-history enumeration. The number of combinations to verify for E2 is reduced from $2^9 - 1$ down to 54.

Answer to RQ 6. The two types of slice minimization techniques are complementary to each other – the heuristic-based static pattern matching technique (S1) reduces the search space with little performance overhead, and the dynamic sub-history enumeration technique (S2) guarantees minimality of the results.

6.4.2 Experiment 9: Comparison with Delta Debugging

In order to answer RQ 7, we compare CSlicerCloud with delta debugging to evaluate the performance of the slice minimization optimization. Delta debugging can be considered a form of semantic slicing with respect to the failure-inducing properties. Therefore, it is natural to apply the same idea for minimizing semantic slices.

We implemented delta debugging within our tool framework to allow a fair end-to-end comparison between the two approaches. Delta debugging follows a divide-and-conquer-style history partition process. In each iteration, a subset of the commits was reverted and if the resulting program passed the tests, the process was continued recursively on the remaining sub-history. Otherwise, a different partition was attempted. We applied both delta debugging and CSlicerCloud on 10 benchmarks for which we have established the ground truth (see Table 6.1). We looked at both the total running time and the number of tests required to reach the minimal solution.

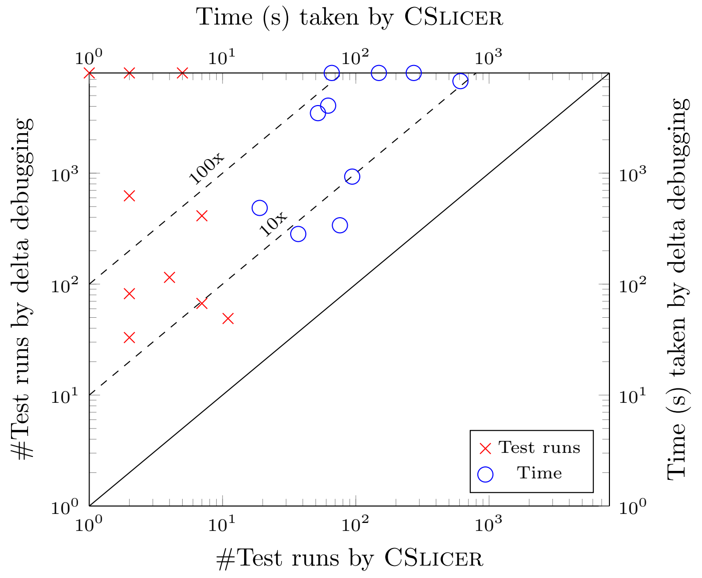


Figure 6.8: End-to-end comparison results of delta debugging and CSlicerCloud.

The detailed results are shown in Figure 6.8. The left and right y-axes represent the number of tests and the total running time required by delta debugging to find a minimal solution, respectively. The bottom and top x-axes represent the number of tests and the total running time required by CSlicerCloud to finish, respectively. All axes use the logarithmic scale. The solid diagonal line represents the cutoff where CSlicerCloud performs the same as delta debugging. The two dashed diagonal lines represent the cutoffs where CSlicerCloud has a 10x and 100x speedup, respectively.

CSlicerCloud performs better than delta debugging for points above the solid diagonal line which was observed for all examples we ran. There were three examples (H1, H2, and M3) where delta debugging could not find a minimal solution within the two-hour time limit (points lying on the top x-axis). On average, CSlicerCloud took only 143.7 seconds to finish while delta debugging took over 2,331.0 seconds (not including the ones where it ran out of time). The main reason behind CSlicerCloud's win is that repeated test executions are rather expensive and Phase 1 of CSlicerCloud can effectively reduce the search space. Moreover, Phase 1 is relatively inexpensive, taking only 37.4% of the total running time. Overall, CSlicerCloud performs constantly better than delta debugging, with the majority of the cases seeing a 10X to 100X speedup. Finally, in cases where both CSlicerCloud and delta debugging finished, they also found the same solutions.

Answer to RQ 7. CSlicerCloud in the minimization mode outperforms delta debugging while maintaining the same slicing quality.

6.5 Summary

In this chapter, we described our experience of implementing a semantic history slicing framework, CSlicerCloud. The flexible architecture of CSlicerCloud enables the integration of many different history slicing algorithms as well as optimizations, and its Web-based interface allows users to access the history slicing service within a browser. The various optimizations on both the slicing quality and performance are shown effective in our experiments.

Chapter 7

History Slicing in Evolution Management – a Study of Feature Location

"Study the past if you would define the future."

Confucius

7.1 Introduction

Feature location techniques aim to locate pieces of code that implement a specific program functionality, also known as a *feature*. These techniques support developers during various maintenance tasks, e.g., locating code of a faulty feature that requires fixing, and are extensively studied in the literature [DRGP13, RC13b]. The techniques are based on static or dynamic program analysis, information retrieval (IR), change set analysis, or some combination of the above.

Identifying features in cloned software product variants is important for a variety of software development tasks. For example, developers often need to share features between variants. That becomes a challenging task as it is often unclear which commits correspond to the particular feature of interest [RKBC12, AJB$^+$14, LRC15]. Refactoring cloned variants into *single-copy* SPL representations also relies on the ability to identify and extract code that implements each feature [RKBC12, RC13a, RCC13, RCC15, AJB$^+$14].

As a step towards addressing these problems, we propose a dynamic technique, called FHistorian, for locating features in software version histories. Our technique differs from other work in that it,

- traces features to historical information about their evolution,
- leverages version histories to improve the accuracy of feature location, and
- is efficient even if the number of available product variants is small.

Being dynamic, FHistorian relies on the availability of a test suite T_f exercising a feature of interest f; such test suites are commonly provided by developers for validating features. Starting from T_f, our technique "slices" the history to identify the commits relevant for f. It also analyzes the slices produced for multiple features $f_1, \ldots, f_n$ in order to identify relationships between these features and build a feature model that represents the extracted information. The generated feature model guarantees that all product variants it describes are well-formed, as it captures runtime dependencies between features.

Example 6

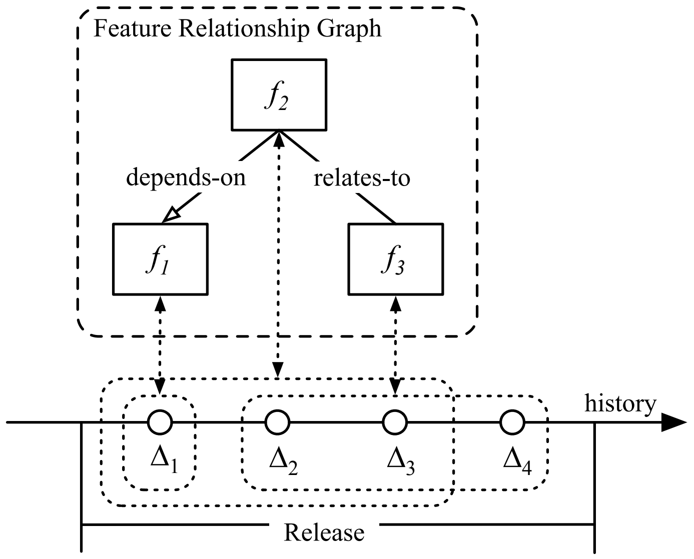


Figure 7.1: A schematic feature relationship graph extracted from `commons-csv v1.3`.

We take a history fragment from the release version 1.3 of an open-source software project `commons-csv` [CSV17] and the simplified commit history is shown in Figure 7.1 as a sequence of commits $\langle \Delta_1, \Delta_2, \Delta_3, \Delta_4 \rangle$. There are three features implemented in this fragment: features "CSV-159" (f_1: add IgnoreCase option for accessing header names), "CSV-179" (f_2: add shortcut method for using first record as header to `CSVFormat`), and "CSV-180" (f_3: add `withHeader(<Class?  extends Enum>)` to `CSVFormat`).

FHistorian identifies a minimal set of commits required by each feature and generates a feature relationship graph depicting the relationships between the features as shown in Figure 7.1. The commits implementing f_1, f_2 and f_3 are $\{\Delta_1\}$, $\{\Delta_1, \Delta_2, \Delta_3\}$, and $\{\Delta_2, \Delta_3, \Delta_4\}$, respectively. Since the commit implementing f_1, namely, Δ_1, is required by f_2 to execute correctly, we say that f_2 *depends on* f_1. Similarly, since f_2 and f_3 both require commits Δ_2 and Δ_3, we say that they *relate to* each other.

The resulting feature model annotated with feature-implementing changes (or *feature changes* for short) 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* relationship indicates connections between features. They often reveal underlying hidden dependencies which are essential across the system.

These relationships indeed exist among the analyzed features. The correct behaviors of CSV-179, "using first record as headers to `CSVFormat`" requires the "IgnoreCase option" (CSV-159) being enabled to produce correct headers. Both CSV-179 and CSV-180 add new functionalities to the `CSVFormat` class and thus are connected to each other.

Contributions

We summarize this chapter's contributions as follows:

- We define FHistorian– a dynamic approach for locating *multiple* features in version histories. In contrast, the semantic history slicing techniques presented in Chapter 4 and 5 locate feature-implementing changes for a single feature at a time.
- We present history-based techniques for building light-weight feature models which represent runtime relationships of features.
- We evaluate the proposed technique on five real-world examples and show its accuracy and effectiveness.

Organization

The rest of this chapter is structured as follows:

- In Section 7.2, we introduce our history-based feature analysis technique FHistorian and describe its feature location and feature relationship inference capabilities.
- In Section 7.3, we present the evaluation of FHistorian on real-world case studies.
- In Section 7.4, we discuss related work.
- Finally, we conclude this chapter in Section 7.5.

7.2 Our Approach

We now present FHistorian, a dynamic feature location technique based on the analysis of commit histories.

7.2.1 Overview

We start by defining the notion of a feature. While there is no universal agreement on what a feature is (and what it is not), we adopt the definition by Kang et al. [KKL+98]:

Definition 13. *(Feature [KKL+98, CR00]). A* feature *is a distinctively identifiable functional abstraction that must be implemented, tested, delivered, and maintained. A feature consists of a label and a short description that identifies its behavior. For conciseness, either the label or the feature description can be dropped when clear from the context.*

As mentioned in Section 2.2.1, we assume that the functionalities of features can be captured by tests and the execution trace of a test is deterministic.

Software version histories are often organized not in terms of features, but as a sequence of incremental development activities, ordered by timestamps. This usually results in a history mixed with changes to multiple features which may or may not be related to each other. Given a piece of history H which is known to implement a set of features F, and each feature $f \in F$ exercised by a test suite T_f, we would like to identify a set of relevant changes for each of the features.

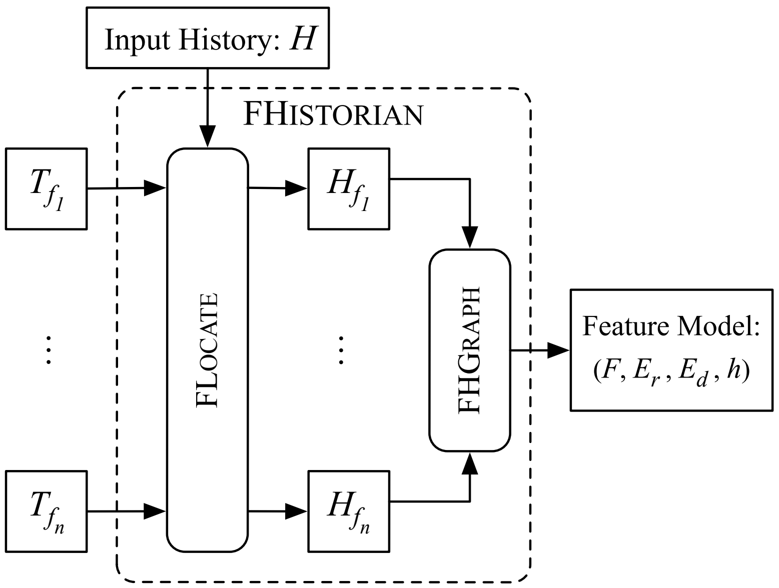


Figure 7.2: Overview of FHistorian architecture.

Figure 7.2 gives an overview of the FHistorian work flow. FHistorian is built on top of the semantic history slicer Definer (Chapter 5). First, we recognize that semantic history slicing, as described in Chapters 4 and 5, is directly applicable for dynamically locating single features, one at a time. An improved version, with hunk-level minimization, is implemented by the FLocate component shown in Figure 7.2. It receives an input history H and a feature test T_f and produces a 1-minimal set of changes relevant to this feature. Then, by consolidating history information of all the target features, the FHGraph component is able to produce a feature model capturing inter-feature relationships such as the runtime dependencies between features. We describe these components below.

7.2.2 Flocate: History Slicing with Hunk-Minimization

The Flocate component of FHistorian is inspired by the existing history slicing technique Definer, extended with *hunk-level minimization.*

Instead of stopping at the minimal history slices at the commit-level, we zoom into individual hunks of commits, to obtain minimal *hunk slices* through hunk-level minimization (see Section 6.3.1). This process yields a set of feature-implementing hunks which are potentially much smaller than the corresponding original commits and contain significantly fewer unrelated changes. In Section 7.3, we empirically show that hunk-level minimization significantly improves the precision of FLocate in locating feature changes.

7.2.3 FHGraph: Inferring Feature Relationships

In addition to locating features in version histories, we also utilize the obtained feature-change information to understand the underlying relationships between features within the same history segment. In particular, we infer two types of feature relationships: *relates-to* and *depends-on*, and represent them in a feature relationship graph. The identified relationships respect well-formedness and functionalities of target features – satisfying feature dependencies is the prerequisite of producing a fully functional product variant. The produced feature model can also assist developers in recognizing interactions between software components by revealing underlying hidden connections.

Feature Relationship Graph

A *feature relationship graph* with respect to a set of features F implemented within a history H is a tuple (F, E_r, E_d, ν), where (F, E_r) is an undirected graph whose nodes are features and edges are relates-to relationships. Similarly, (F, E_d) is a directed graph for depends-on relationships. $\nu : F \mapsto \mathcal{H}$ is a map from features to feature changes.

Relates-To Relationship. We say that a feature f_1 *relates to* another feature f_2 if they both rely on the presence of the same set of non-empty changes $H_{(f_1,f_2)} \subseteq H$ in order to function correctly. Formally, we write $f_1 \leftrightarrow f_2$ when

$$\forall H' \subseteq H \cdot (H' \models T_{f_1} \wedge H' \models T_{f_2}) \Rightarrow H_{(f_1,f_2)} \subseteq H', (H_{(f_1,f_2)} \neq \emptyset).$$

Depends-On Relationship. Similarly, a feature f_1 *depends on* another feature f_2 if f_1 functions correctly only when f_2 does so as well. More formally, we write $f_1 \rightarrow f_2$ if

$$\forall H' \subseteq H \cdot (H' \not\models T_{f_2}) \Rightarrow (H' \not\models T_{f_1}).$$

More generally, a product variant defined by the history H_v for features $F_v = \{f_1, \ldots, f_n\}$ is semantics-preserving (i.e., $\bigwedge_{f_i \in F_v} H_v \models T_{f_i}$) only when F_v is closed under their feature dependencies. In other words, satisfying all of the dependencies is the prerequisite for the product variant to behave as expected.

	IDs	Descriptions
Feature Changes	δ_1	`i:  int f1(){return 1;}`
	...	...
	δ_2	`j:  int f2(){return f1()+1;}`
	...	...
	δ_3	`k:  int f3(){return f1()-1;}`
Feature Tests	T_{f_1}	`f1()==1`
	T_{f_2}	`f2()==2`
	T_{f_3}	`f3()==0`

Figure 7.3: An example illustrating feature relationships.

Example 7. Figure 7.3 shows an example of a history $H = \langle \delta_1, \delta_2, \delta_3 \rangle$ containing changes adding three new features in sequence: $F = \{f_1, f_2, f_3\}$. During the given history, three lines of code are inserted one after another – Line i, Line j and then Line k – each reflecting a feature implementation change δ_1, δ_2, and δ_3, respectively. The corresponding feature tests $\{T_{f_1}, T_{f_2}, T_{f_3}\}$ are shown at the bottom. For example, the feature f_1 is implemented as a function `f1()` which is expected to return an integer 1.

It is easy to see that f_2 depends on f_1 and f_3 depends on f_1, since both functions `f2()` and `f3()` require the definition of `f1()` introduced in the change δ_1. Likewise, we also have that f_2 is related to f_3, and their witness change $H_{(f_2,f_3)}$ is δ_1.

Discovering Feature Relationships

To infer relationships for the target features, we consolidate all the history slicing results returned by FLocate and determine pairwise feature relationships by comparing their minimal semantics-preserving slices.

For example, when the semantic slice of f_1 is subsumed by that of f_2, i.e., $H_{f_1} \subseteq H_{f_2}$, we would say f_2 depends on f_1 and also factor out H_{f_1} from the semantic slice of f_2. The resulting feature changes identified for f_2 would be $H_{f_2} \setminus H_{f_1}$. More formally, the algorithm for inferring feature relationships is based on the following theorem.

Theorem 5. *Suppose the minimal semantics-preserving slices for f_1 and f_2 are both unique in H, denoted by H_{f_1} and H_{f_2}, respectively. We have $(f_1 \leftrightarrow f_2) \Leftrightarrow (H_{f_1} \cap H_{f_2}) \neq \emptyset$, and $(f_2 \rightarrow f_1) \Leftrightarrow H_{f_1} \subseteq H_{f_2}$.*

Proof. The key for proving this theorem lies in realizing that if the minimal semantics-preserving slice H_f for a feature f is unique, then H_f is essential for passing the feature test T_f, i.e., $\forall H' \subseteq H \cdot H' \models T_f \Rightarrow H_f \subseteq H'$. Hence, $H_{f_1} \cap H_{f_2}$ serves as the witness for $f_1 \leftrightarrow f_2$, i.e., $H_{(f_1,f_2)} = H_{f_1} \cap H_{f_2}$. Similarly, when $H_{f_1} \subseteq H_{f_2}$, H_{f_1} is essential for both f_1 and f_2. □

Algorithm 6 An algorithm for constructing a history-based feature relationship graph.

```
1:  procedure FHGraph($F, H$)
2:      $E_r, E_d, \nu \leftarrow \emptyset, \emptyset, \emptyset$
3:      for $f \in F$ do
4:          $H_f \leftarrow$ FLocate($H, T_f$)                          ▷ get minimal slices
5:          $\nu(f) \leftarrow H_f$
6:      end for
7:      for $(f_1, f_2) \in F \times F$ s.t. $f_1 \neq f_2$ do
8:          if $H_{f_1} \cap H_{f_2} = \emptyset$ then continue
9:          if $H_{f_1} \subseteq H_{f_2}$ then
10:             $E_d \leftarrow E_d \cup (f_2 \rightarrow f_1)$                 ▷ depends-on
11:             $\nu(f_2) \leftarrow H_{f_2} \setminus H_{f_1}$                 ▷ factor out $H_{f_1}$ from $H_{f_2}$
12:         else if $H_{f_2} \not\subseteq H_{f_1}$ then
13:             $E_r \leftarrow E_r \cup (f_1 \leftrightarrow f_2)$                 ▷ relates-to
14:     end for
15:     return $(F, E_r, E_d, \nu)$
16: end procedure
```

With the assumption of unique minimal semantics-preserving slices, an algorithm for constructing feature relationship graph is given in Figure 6. The procedure FHGraph receives a set of features F and the history H implementing them. It updates two edge sets, E_r and E_d, for the relates-to and depends-on relationships, respectively. It also maintains a map h which stores the final feature location results for each feature.

First, we store the minimal semantics-preserving slices returned by FLocate for all features (Line 4). The algorithm then iterates through all feature pairs (Line 7 – 14). For each pair of minimal slices H_{f_1} and H_{f_2}, if H_{f_1} is subsumed by H_{f_2}, then we create a depends-on edge $f_2 \rightarrow f_1$ and factor out H_{f_1} from H_{f_2} and store it into $\nu(f_2)$ (Line 10 and 11). If there is no depends-on relationship between f_1 and f_2, then a relates-to edge is constructed instead (Line 13). The procedure returns a feature relationship graph (F, E_r, E_d, ν) when it terminates.

7.3 Evaluation

In this section, we present the empirical evaluation of our approach on real-world software systems. Our goal is to have a better understanding of the capability of our history-based feature analysis technique by answering the following research questions:

RQ 8 How accurate is the feature location performed by FHistorian?

RQ 9 How accurate are the feature relationships inferred by FHistorian?

We implemented FHistorian on top of Definer (Chapter 5), and used the interactive mode of the Git `add` command to automatically split commits into hunks. We also generated feature relation graphs represented using the DOT graph format. Our prototype implementation is available at: `https://bitbucket.org/liyistc/gitslice`.

7.3.1 Subjects

To effectively evaluate FHistorian, we need access to project source code, test cases, and commit histories to run feature analysis, as well as adequate feature annotations to determine its effectiveness. In particular, to perform feature location within a history segment, FHistorian requires a predefined set of features, known to be implemented in the history period, along with test cases.

To find suitable evaluation subjects, we looked for complete histories between two software releases and referred to release notes to determine the newly implemented feature set. We selected experimental subjects from a combination of recently published history analysis datasets (Appendix A) which are well-documented and organized, and ended up with five releases accompanied by comprehensive release notes and feature annotations. All selected software projects use JIRA [JIR17] as their issue tracking system. In JIRA, each feature has a unique developer-assigned ID, referred to as the "issue key", which is associated with an issue report recording detailed information about the feature. A JIRA issue key is a string with the format "ABC-123", where "ABC" stands for the name of the project containing this feature, and "123" is a unique ID. Developers label commits with issue keys to indicate the purpose of the changes, which enables us to determine which feature models the developers had in mind.

Table 7.1: Statistics of the experimental subjects used in the evaluation of FHistorian.

Project & Release	#C	#F	LOC	#Issue	Features #New	Features #Tested
`commons-csv v1.3`	79	28	2353	12	7	4
`commons-compress v1.13`	148	144	6650	13	7	6
`commons-io v1.4`	140	140	8607	24	18	9
`commons-io v2.2`	136	182	7328	24	15	7
`commons-lang v3.4`	262	146	8817	63	17	10

The set of features implemented during the release histories was determined from the release notes. In all the release notes that we analyzed, newly implemented functionalities are organized as issues including new features, tasks, bug fixes, improvements, etc. For the purpose of our experiments, we focused on releases that had at least four implemented and tested features. The resulting subjects are summarized in Table 7.1. Each row represents a history segment for a particular release. Column "Project & Release" designates the name of the project from which the releases are chosen, followed by their version number. Columns "#C", "#F", "LOC" represent the number of commits, the number of files modified, and the number of lines of code changed during the release histories, respectively. Column "#Issue" represents the number of all issues reported. Columns "#New" and "#Tested" represent the number of all new features and those with associated tests cases. For example, the second row of Table 7.1 shows that during the development of version 1.13 of the project `commons-compress`, there were 148 commits, 144 files were modified and 6650 lines of code have been changed. Developers created 13 issues, seven of which were documented as new features; among them, six were accompanied by test cases capturing the expected feature behaviors.

7.3.2 Experiment 10: Precision of Feature Location

Experiment 10 aims to address RQ 8 by running FHistorian on the selected subjects and comparing the result with developer annotations.

Methodology

We compared the feature location results of FHistorian with developers' feature annotations found in the commit logs. For each new feature found in the release notes, we used the assigned JIRA issue key to map the feature with the commits considered as the feature implementations by the developers. We then ran FHistorian on the whole feature set, taking feature relations into consideration, to compute relevant changes for each feature. Finally, we repeated the same experiments with hunk-level minimization disabled to decide the effectiveness of our optimizations on commit-level history slicing.

Results

The studies were conducted on a desktop computer running Linux with an Intel i7 3.4GHz processor and 16GB of RAM. It took on average 4,062 seconds for Definer to obtain a 1-minimal semantic slice for each feature. The hunk minimization on the resulting slice took on average 3,740 seconds per feature.

Table 7.2 lists the feature location results of FHistorian, comparing them with the developers' feature annotations. Each row in the table shows results for a particular feature, identified by the feature key. Column "Releases" lists the release histories being analyzed. Columns "#Labeled" and "#Found" show the number of commits labeled by the developers and identified by FHistorian, respectively. We also list the differences between their results in the last two columns – Column "#FN" shows the number of commits labeled by developers but not found by us and vice versa for Column "#FP". For instance, the developers annotated one commit for feature "CSV-159" and FHistorian found the same commit. However, for "CSV-175", three commits were annotated by the developers and FHistorian found one of them with five extra commits and missed the other two.

For 15 out of 36 features, FHistorian's results match perfectly with the developers' annotations. To understand the differences in the rest of the cases, we analyzed all of the FHistorian's false positives and false negatives.

False Positives

FHistorian includes not only conceptually essential changes but also peripheral changes to guarantee the executability of its produced feature models. When committing and labeling feature changes, developers

Table 7.2: Feature location results of FHistorian compared with developer annotations.

Releases	Feature Keys	#Labeled	#Found	#FN	#FP
`csv v1.3`	CSV-159	1	1	0	0
	CSV-175	3	6	2	5
	CSV-179	1	8	0	7
	CSV-180	1	8	0	7
`compress v1.13`	COMPRESS-327	10	17	2	9
	COMPRESS-368	6	5	3	2
	COMPRESS-369	2	5	1	4
	COMPRESS-373	1	7	0	6
	COMPRESS-374	1	4	0	3
	COMPRESS-375	2	1	1	0
`io v1.4`	IO-126	1	1	0	0
	IO-129	2	1	1	0
	IO-130	1	1	0	0
	IO-135	2	1	1	0
	IO-138	1	1	0	0
	IO-144	1	1	0	0
	IO-145	1	1	0	0
	IO-148	4	1	3	0
	IO-153	1	1	0	0
`io v2.2`	IO-173	1	1	0	0
	IO-275	1	1	0	0
	IO-288	3	1	2	0
	IO-290	1	1	0	0
	IO-291	3	2	1	0
	IO-297	1	1	0	0
	IO-305	1	1	0	0
`lang v3.4`	LANG-536	1	3	0	2
	LANG-883	1	1	0	0
	LANG-993	1	1	0	0
	LANG-999	1	1	0	0
	LANG-1015	1	1	0	0
	LANG-1021	1	2	0	1
	LANG-1033	1	1	0	0
	LANG-1080	1	1	0	0
	LANG-1082	1	1	0	0
	LANG-1093	2	1	1	0

often overlooked preexisting changes which support the compilation and execution of the features. For example, FHistorian considered commit `#f8e09945` as necessary for the feature COMPRESS-369 but it is not labeled by the developers. We inspected the commit, finding it to be a bug fix updating the configuration file to use a newer Java JDK. The target feature code cannot be compiled or executed when ignoring this commit, and thus it is essential.

The second reason why FHistorian detects more commits is that it also takes *hunk dependencies* (see Section 4.3.3) specific to text-based version control systems into consideration. That is, FHistorian includes additional commits providing a necessary context for the application of the essential commits. We verified that all the feature changes found by FHistorian were minimal, i.e., they could not be further reduced and yet pass the feature tests. Thus, all commits found by FHistorian but not labeled by the developers were required for the correct execution of feature tests.

False Negatives

On the other hand, FHistorian occasionally missed commits labeled by the developers (28% missed). We manually inspected each missed commit and summarize the most common reasons below.

Some commits missed by FHistorian contained only changes which did not affect feature execution. For example, developers occasionally created separate commits that updated the release note file documenting addition of a new feature, and then labeled them as part of the feature. There were also commits labeled as feature implementations which only updated Javadocs or performed refactoring. FHistorian also missed commits which performed minor optimizations which were labeled as part of the feature but the associated feature tests were not updated to capture the modified behaviors.

Effectiveness of Hunk-level Minimization

The comparison of the feature location precision results of FHistorian with and without the hunk-level minimization (i.e., staying at the commit level) are shown in Figure 7.4. On average, hunk-level minimization improved FHistorian's precision 3.47 times. In particular, for releases `commons-io v1.4` and `commons-io v2.2`, hunk-level minimization yielded a 16X improvement in precision (from 5.21% to 81.8% and from 6.32% to 100%, respectively).

Answer to RQ 8. FHistorian precisely identifies commits that implement a specific feature and required for the feature execution. Its feature location results coincide with the developer annotations with regard to essential feature implementation changes, while they differ from the annotations when considering changes that do not affect the test execution. The commit-level optimizations for history slicing are effective in improving the precision of feature location.

7.3.3 Experiment 11: Accuracy of Feature Relationships

Experiment 11 is designed to answer RQ 9 by running FHistorian on all the new features added in the same release, which have associated test suites. We are interested in feature relationships discovered by FHistorian.

Methodology

In this experiment, we evaluated the accuracy of feature relationships inferred by FHistorian through code inspections and qualitative studies of feature documentation. For each subject release, we ran FHistorian with the same feature set used in answering RQ1 to generate a feature relationship graph. We then used additional feature annotations extracted from release notes to refine the feature relationships. Finally, we analyzed each of the inferred feature relationships and verified it either by internal code inspection or external evidence such as log messages and issue links on JIRA issue pages.

Results

To simplify the discussion, we categorize the five releases studied into two groups. In one group, namely, `commons-io v1.4, v2.2` and `commons-lang v3.4`, the feature relationships are relatively sparse and only depends-on relationships are observed. In the other group, namely, `commons-csv v1.3` and `commons-compress v1.13`, the feature relationships are more complex and the relates-to relationships between tested features often reveal "hidden" untested features. We show that with the additional

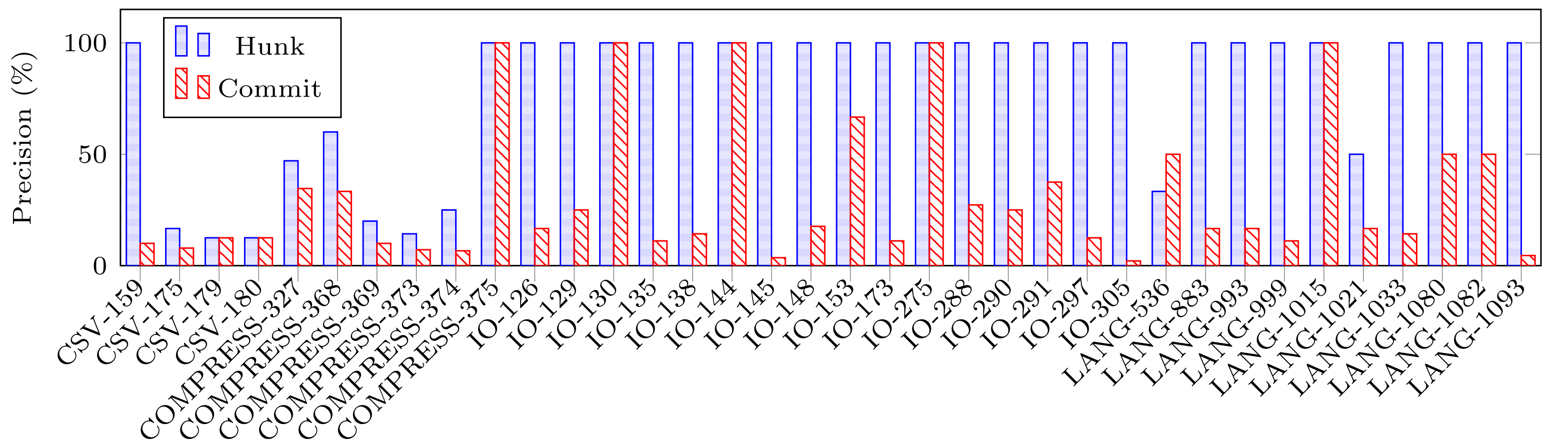


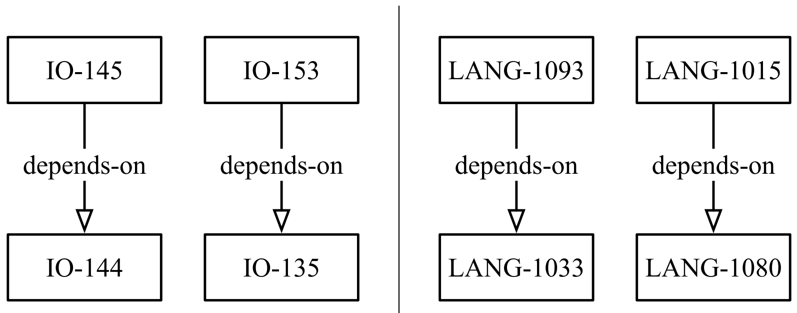
Figure 7.4: Precision of FHistorian's feature location with and without hunk-level minimization.

knowledge about the hidden features, the relates-to relationships can be refined and reflect more accurate relationships among analyzed features.

Case Study 1: `commons-io` and `commons-lang`

No feature relationships were observed for release v2.2 of `commons-io`. All analyzed features in this release can be independently executed using non-overlapping commits.

There are two depends-on edges detected in release v1.4 of `commons-io`: from IO-153 to IO-135 and from IO-145 to IO-144 (see Figure 7.5a, where feature nodes without relationships are omitted).



(a) `commons-io v1.4` (b) `commons-lang v3.4`

Figure 7.5: Feature relationships for `commons-io` and `commons-lang`.

Similarly, Figure 7.5b illustrates the relationship between the features in release v3.4 of `commons-lang`. This example also has two depends-on edges: from LANG-1093 to LANG-1033, and from LANG-1015 to LANG-1080.

To verify the feature relationship found by FHistorian, we report evidence observed from log messages and source code.

(1) IO-144 → IO-145, IO-153 → IO-135, LANG-1093 → LANG-1033. The three depends-on edges were verified by inspecting contents of the commits. For example, feature IO-144 is implemented by a single commit `#db3e834e` with the commit message "add a compare method". We inspected the code changes in this commit and found that it creates a special "checkCompareTo()" method for comparing strings with the ability to adjust case sensitivity. The commit message of IO-145 (`#55dfa6eb`) is "add new package of file comparator implementations". This commit creates and modifies multiple file comparator classes. Three of these, namely, "ExtensionFileComparator", "NameFileComparator", and "PathFileComparator", rely on the "checkCompareTo()" method for file name comparison. This observation confirms that feature IO-145 depends on IO-144 for its implementation, which in turn supports the identified feature relationship. The other two edges are similar – one feature provides a utility function which is then used by the other feature, thus creating a feature dependency between the two.

(2) LANG-1015 → LANG-1080. FHistorian's results indicate that feature LANG-1015 is implemented by two commits, one of which labeled as LANG-1015 and the other as LANG-1080. By inspecting the commits, we concluded that LANG-1015 depends on LANG-1080 due to the hunk dependency. Patching the former without the latter using Git would result in a conflict because the two commits modify adjacent lines in the same file. This dependency can of course be ignored if a language-aware version control system (e.g., SemanticMerge [Sem16]) is used instead.

Case study 2: `commons-csv` and `commons-compress`

In this case study, we first show the feature models produced by FHistorian using tested features. We then show how additional feature annotations extracted from release notes and log messages can be used to refine these feature models (e.g., with explicating "hidden features").

(1) commons-csv. Figure 7.6 (top) illustrates the feature relationship graph originally obtained by FHistorian in the release v1.3 of `commons-csv`. In this example, there are three depends-on edges, indicating that CSV-175 → CSV-159, CSV-179 → CSV-159, and CSV-180 → CSV-159, respectively. There are also three relates-to edges, showing that CSV-175, CSV-179, and CSV-180 are related to each other.

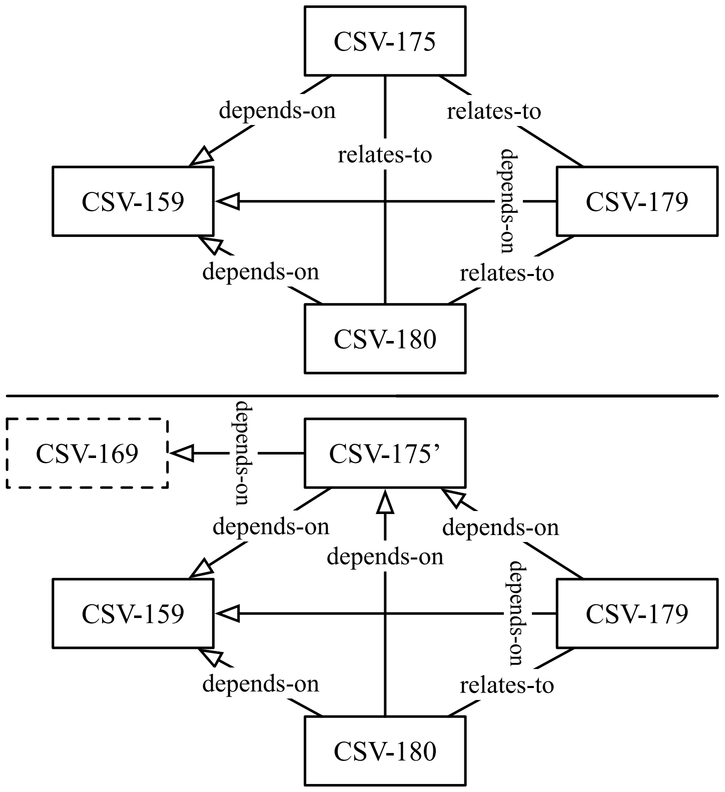


Figure 7.6: Feature relationships for `commons-csv 1.3`.

To obtain a more precise relationship between these features, we took into consideration bug fix annotations in the release notes and discovered that feature CSV-175's member commits can be separated into two groups, with one implementing the feature's functionality, and the other, labeled as CSV-169, contributing to a bug-fix. The bug-fix does not belong to CSV-175 technically. But since there is no test case associated with it, FHistorian could not distinguish the bug-fix commit from the feature-implementing commit.

Using this obtained information, we created a new feature node, CSV-169, to represent the "hidden" bug-fix (in dashed box). We use node CSV-175' to represent the rest of the commits that originally belonged to CSV-175. With the updated feature set, FHistorian computed a newly refined relationship graph shown in Figure 7.6 (bot). Since the commits of CSV-175' are fully subsumed by both CSV-179 and CSV-180, Two original relates-to relationships, CSV-179 ↔ CSV-175 and CSV-180 ↔ CSV-175 can be refined into stronger depends-on relationships, namely, CSV-179 → CSV-175' and CSV-180 → CSV-175'.

The refined feature relationships better reflect the actual situation. Some evidences confirming these relationships can be observed from the source code and commit messages. For example, CSV-179 and CSV-180 both rely on the "ignoreHeaderCase" option, created by CSV-159, to generate CSV file headers. This verified the inferred depends-on relationships CSV-179 → CSV-159 and CSV-180 → CSV-159.

(2) commons-compress v1.13. Figure 7.7 (top) illustrates the feature relationships in release v1.13 of `commons-compress`. From FHistorian's result, we can determine that features COMPRESS-327, COMPRESS-368, COMPRESS-369, COMPRESS-373, and COMPRESS-374 all relate to each other. However, the underlying reasons for these relationships were unclear without additional knowledge about the software project.

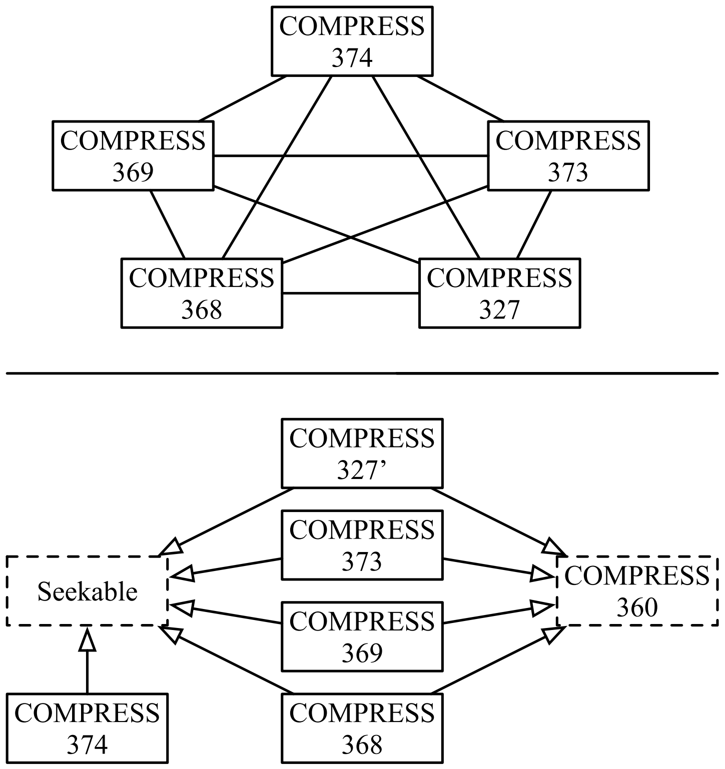


Figure 7.7: Feature relationships for `commons-compress v1.13`.

We inspected the relevant commits, aiming to build a more precise feature relationship graph. We discovered that all features rely on a shared commit, `#7e35f57`, labeled as COMPRESS-327. This shared commit upgrades a basic component – it re-implements the output stream of Zip format using a new class named "SeekableByteChannel", replacing an old class "RandomAccessFile". This upgraded component is widely used as a basis for many other functionalities related to stream compressors. Therefore, this commit affects all the five features mentioned earlier.

Another commit `#f8e09945` is also shared among features. It contributes to the implementation of four features: COMPRESS-327, 368, 369, and 373. Upon further investigation, we determined that it is another bug-fix commit, labeled by the developers as COMPRESS-360, which updates the project's minimum JDK version requirement from 1.6 to 1.7. The change was made in the project configuration file and without this change, the other four features failed to compile.

We separated the shared commits from COMPRESS-327 to create individual nodes for them in the feature relationship graph ("Seekable" and "COMPRESS-360" in dashed boxes). As a result, the refined feature relationship graph is shown in Figure 7.7 (bot). The new graph reveals multiple new feature relationships. First, it now shows that COMPRESS-327, COMPRESS-368, COMPRESS-369, and COMPRESS-373 all depend on the hidden nodes "Seekable" and "COMPRESS-360". In addition, COMPRESS-374 also depends on "Seekable". The original relates-to edges can be trivially inferred from the current graph: two nodes depending on the same node are automatically connected by related-to. We omit those edges in Figure 7.7 (bot).

We now discuss evidences in support of the found feature relationships. We found direct evidence provided by the developers for the edge COMPRESS-368 $\leftrightarrow$ COMPRESS-369. Developers explicitly labeled the relationship between COMPRESS-368 and COMPRESS-369 with a JIRA issue link, "is related to COMPRESS-369 (allow archiver extensions through a standard JRE ServiceLoader)" on the issue description page of COMPRESS-368.

Answer to RQ 9. Feature relationships inferred by FHistorian are accurate. The depends-on relationships reflect runtime dependencies between features. They are essential for ensuring well-formedness and correct execution of the product variants constructed from the target features. The relates-to relationships are useful in revealing underlying connections between features and can be further refined into stronger depends-on relationships using additional project expertise such as issue tracking tickets, developer conversations, log messages, etc.

7.3.4 Threats to Validity

While we selected different projects and attempted to cover different scenarios, our results may not be sufficiently representative. Furthermore, the projects that we selected for evaluation have complete change logs and release notes. While our feature location technique produces encouraging results on our experimental subjects, it is not always applicable to projects that are not well-managed. In the absence of documentation of the release histories and the corresponding feature information, expert insights are required for FHistorian to achieve comparably good results.

Due to the absence of adequate feature documentation, it is not always possible to verify the feature relations obtained by FHistorian rigorously with developers' conceptual models. For example, we cannot be certain whether all of the relationships between the features have been generated. Our results were therefore confirmed by multiple indirect evidences such as commit messages, contents of code changes, etc.

7.4 Related Work

We discuss related work in four areas, namely, dynamic feature location, feature location in version histories, feature location for software product lines, and extracting feature models from code.

7.4.1 Dynamic Feature Location

Dynamic feature location techniques rely on program execution for identifying source code that corresponds to a feature of interest. More than 10 such techniques are reviewed in [RC13b, DRGP13]. The earliest dynamic feature location technique which also became a foundation of future approaches is *software reconnaissance* [WS95]. It compares execution traces obtained by exercising the feature of interest to those obtained when the feature is inactive. The technique runs a set of test cases that invoke each feature and extracts components (code statements or methods) executed by each test case. For each feature, it then identifies the (1) indispensably involved components – executed by all test cases of the feature, (2) potentially involved components – executed by at least one test case of the feature, and (3) uniquely involved components – executed by at least one test case of the feature and not executed by any test case of the other features. It also extracts the set of common components executed for all features.

Several later approaches extended this work by involving static code analysis, information retrieval, and other techniques to further prune the feature execution traces and improve the accuracy for feature detection, allowing them to operate on a single trace rather than on multiple traces corresponding to multiple features [LMPR07, RDP10]. FHistorian also relies on the presence of test cases to perform feature location. Yet, it detects features in change histories rather than in the "final" version of the program, thus assisting in tasks such as porting features and their histories across multiple branches in version control systems. It also uses information about change histories to improve the accuracy of feature location and does not require tests of multiple features in order to operate.

7.4.2 Feature Location in Version Histories

In CVSSearch [CCW+01], a feature is specified as a text query. The technique uses the CVS "diff" command to examine changes between subsequent commits and associates each line of code changed in a commit with its corresponding commit message. It then retrieves all lines that match the input query, i.e., either the line itself or its associated message containing at least one word from the query. The results are scored and ranked so that files with the highest number of matches appear first. Unlike CVSSearch, our technique does not rely on textual similarity but rather analyzes semantics of changes and extracts executable feature implementations.

7.4.3 Feature Location for SPLs

Each of the existing feature location techniques can be used for detecting features in products of a product line by treating these products as singular independent entities. Yet, several techniques that consider commonalities and differences in SPL products have recently emerged [XXJ12, ZFdSZ12, ASH+13b, ASH+13a, LAG+14, LLE13, TKW+13, LLHE16]. Most such techniques are based on intersecting code of multiple product variants in order to identify code fragments shared by variants with a particular feature. For example, Xue et al. [XXJ12] use information about version differencing to further improve the accuracy of information retrieval in multiple products. Linsbauer et al. [LLE13, LLHE16] present a technique for deriving the traceability between features and code in product variants by matching code overlaps and feature overlaps. Moreover, this technique also identifies code that depends on the combination of features present in a product variant thus dealing with feature dependencies and interactions. While the above interaction-based techniques operate statically and are effective when a large number of product variants are available, our approach is dynamic and does not rely on the presence of a large set of variants to be effective. Moreover, it is also able to distinguish between features that always appear together in all product variants – a clear limitation of the intersection-based techniques.

7.4.4 Extracting Feature Models

Several approaches focus on extracting constraints between features of multiple variants [RPK11, HLHE11, AHS+14, SSS14, NBKC14, ALHL+15] or on building a desirable feature model when such constraints are given [SLB+11, CW07, ACSW12]. For example, Assunção et al. [ALHL+15] extend the intersection-based feature location technique [LLE13] with an approach to identify dependencies between features by looking at shared/exclusive code fragments. The above approaches mostly consider product and feature combinations, without inspecting semantic dependencies between code artifacts. Moreover, they rely on the availability of multiple product variants while FHistorian does not make such an assumption and is also able to identify dependencies between features of a single variant.

Nadi et al. [NBKC14] focus on inferring constraints between features implemented in C by statically identifying potential preprocessor, parser, linker, and type errors. Instead, our approach is dynamic and precisely identifies all runtime dependencies between the executed features.

7.5 Summary

In this chapter, we presented a dynamic feature location technique FHistorian. The technique works by analyzing version histories, taking into account feature release information and developer-committed tests demonstrating the new feature, to precisely extract feature-related changes. It also produces models representing runtime relationships between features. Our cases studies on multiple features of five versions of four real-world software projects show that FHistorian can locate features effectively.

Chapter 8

Conclusion and Future Work

We have arrived at the end of this dissertation. In Section 8.1, we recap the highlights of this dissertation and in Section 8.2 we discuss limitations of our approach and suggest possible future paths to address them.

8.1 Dissertation Summary

Software Configuration Management (SCM) systems are widely used in hosting development artifacts and assisting software maintenance. Yet, many evolution management tasks which require high-level understanding of the semantics of program changes remain challenging to software developers. In this dissertation, we addressed a number of such challenges through the semantic history slicing approach. By bringing program semantics into history analysis, we fill the gap between local changes and global system properties and provide developers with a more powerful support in evolution management. Our conceptual and algorithmic contributions are demonstrated with a family of automated history slicing techniques which analyze the semantics of historical changes and assist developers in many everyday practical settings.

Subsequent to Chapter 2 where we introduced terminology and formalized the notations that we use, Chapter 3 presented a prestudy on the commonly followed evolution management practices in open-source development environment. The "cherry-picking" feature provided by traditional SCM tools, such as Git, allows developers to move changes as patches, and reorganize them as needed. However, without a good understanding of the semantics and relationships between changes, manual "cherry-picking" often ends up in failures. On the other hand, developers put a significant amount of energy in producing test cases for every functionality they implement, which is also often a strict requirement in well-organized software projects. In the absence of concrete and precise specifications of program semantics, test cases can be used as semantic indicators for the associated software functionalities instead: the passing of a test indicates the presence of the corresponding functionality. Inspired by this, we proposed the concept of semantic history slicing, where test cases are used as slicing criteria to select changes from version histories. The selected changes are known as a semantics-preserving slice of the original change history and they preserve the semantic properties manifested by the tests. We then formalized the notion of semantics-preserving slice and minimal semantics-preserving slice, and outlined the general approaches for computing them.

In Chapter 4, we presented a static history slicing technique, CSlicer, which is based on change dependency analysis. CSlicer compiles and runs the target tests once, and conservatively identifies atomic changes in the given input history which contribute to the runtime behaviors of the tests. Then it performs static dependency analysis over change sets to compute a set of changes which are essential for ensuring compilation correctness of the functionality of interest. Finally, the collected set of atomic changes is mapped back to the SCM specific commits in the original change history, and the hunk dependency analysis over commits helps avoid merge conflicts that can occur when commits are cherry-picked in text-based SCM systems. Being conservative, CSlicer can sometimes label unnecessary changes as required resulting in non-minimal history slices. Our experimental evaluation showed that the static history slicing technique is effective in identifying semantics-preserving history slices and enables a wide range of applications in evolution management.

In Chapter 5, we presented a dynamic history slicing technique, Definer, which can achieve better precision than its static counterpart through the dynamic delta refinement process. Definer executes target tests multiple times and directly observes the test results while attempting to reach a minimal semantics-preserving history slices iteratively. While similar to delta debugging in terms of the divide-and-conquer-style history partitioning, Definer employs a delta refinement mechanism to estimate the significance of changes with respect to the target tests and guide subsequent history partitions for faster convergence. The history slices produced by Definer are guaranteed to be 1-minimal – removing any single commit from the history slice breaks the desired semantic properties. We demonstrated the applications of Definer in evolution management tasks that require minimal semantics-preserving history slices, such as creating logically focused pull requests for code review.

We unified different history slicing algorithms and discussed our experience developing a semantic history slicing framework in Chapter 6. The Web-based history slicing framework, CSlicerCloud, provides users with a flexible and intuitive way to interact with the underlying history slicing engines and implements a number of important optimizations to improve both the slicing quality and efficiency. Through experimentation, we evaluated the effectiveness of combining the cheaper but imprecise static slicing technique with the dynamic slice minimization techniques, and found that their synergy enables a combined approach which is both precise and efficient.

In Chapter 7, we presented a detailed study on applying semantic history slicing in locating features from cloned software products. We built a feature location technique FHistorian on top of Definer. FHistorian is able to identify feature changes from release histories containing multiple feature implementations. It uses feature tests as slicing criteria to compute corresponding history slices for each feature and then generates a light-weight feature model capturing relationships between features. The feature relationships indicate dependencies and connections between features which are essential for constructing valid products. We evaluated the technique on a number of real-world case studies and compared our results with developer-specified feature annotations. We concluded that, when available, historical information of software changes can lead to a precise identification of features in existing software artifacts.

8.2 Future Research Directions

Despite all the advances made in this dissertation, many related problems remain open. We divide the discussion of future research directions into short- and long-term goals. The former describe “immediate” improvements for the techniques developed in this dissertation. The latter focus on “larger” and longer-term goals that contribute to the field.

8.2.1 Short-Term Goals

Deep Integration with Development Tool Chains. While we have developed a family of automated semantic history slicing techniques, all of them are standalone tools which require additional configurations and learning for new users. To further improve tool usability and reach out to a broader user base, an immediate next step is to deeply integrate the history slicing techniques with the commonly used development tool chains. As a first step, we have developed a Maven [Mvn15] plugin, which wraps our history slicing engines as Maven *build goals*. This allows the history slicing engines to be configured and triggered through a familiar user interface by sharing information, such as compilation and testing parameters, directly with the build system. Further, the history slicing process can be activated within one of the Maven build life cycles without altering original development practices.

Integration with popular version control systems, such as Git and Mercurial, is also a promising direction. This will make direct access to new evolution management features, including semantics-aware cherry-picking and automated patch creation, possible within the current version control tools. The results of those operations can then be reflected directly in the repository without additional effort.

Applications in Bug Localization. We would also like to explore the possibility of applying semantic history slicing to assist debugging and fault localization. As mentioned in Section 5.6.2, debugging can be considered a special case of semantic history slicing: given a series of changes which cause a program failure, the goal of debugging is to locate the minimal cause of the failure, or in other words, the changes which preserve the failure-inducing property. A natural extension to the iterative minimization-based debugging approaches (e.g., delta debugging) is to introduce significance learning and delta refinement (Chapter 5) into the input partition process for faster convergence.

Traditional bug localization techniques [SLKP13, WL16], based on information retrieval, take a bug report as input and identify source code files/methods that need to be fixed. It is possible to extend current bug localization solutions to identify problematic commits with semantic history slicing, if test cases demonstrating the faulty behaviors can be provided. In that case, we would be able to automatically locate the links between bug reports and culprit changes or commits in version histories. This would provide additional traceability to the bug localization results and enable more accurate issue management.

Expanding the Boundaries of Change Semantics. The change semantics used in history slicing is not limited to the one that is adopted in this dissertation. The semantics of a history slice can be generalized from pure functional correctness to any measurable properties of programs. Therefore, the slicing criteria used in history slicing need not be defined only by observable test results. In fact, many different instantiations of the more general semantic history slicing framework are possible, according to specific use case scenarios.

An example of such instantiations is to create a semantic history slice by analyzing texts in commit logs with natural language semantics. History slicing with this change semantics corresponds to *commit searching*, such as the “grep” feature provided by the “`git log`” command. For software projects where a particular logging format is used, semantic history slicing can also be used to perform structured search via information retrieval and comprehension [SJ12, CFSM16].

Leveraging the Social Aspect of Software Development. Software development is inherently a social process. For example, the first step to address an issue report or a pull request is to start a conversation between the stakeholders of the corresponding software project. The developer conversations often convey useful knowledge and expertise about the project at hand. As such, we aim to enhance FHistorian (Chapter 7) with information retrieval-based feature location techniques [DRP13, ZLX+16] for extracting expert feature-related knowledge from evolution artifacts such as developer conversations, log messages, and documentation. With more complete information about features, our technique can produce potentially more accurate feature models.

8.2.2 Long-Term Goals

Verifying Change Compatibility. Different parts of software are often created by different teams or even different organizations, making their integration and quality assurance time-consuming and difficult. Moreover, changes in parts of software may not be appropriately communicated to the rest of the system which may have a major impact on the overall system behavior, rendering previous testing and quality assurance efforts obsolete, and potentially creating major problems.

In many industrial domains, such as automotive and financial industries, parts of the software are typically delivered by external subcontractors and need to be put together and validated by the original manufacturers. Such cases bring with them a few additional problems, making the situation even more dire: (1) the subcontractor code may not be available for inspection or other static analyses; (2) the size of external software components present is extremely large; (3) the entire code base is expected to support a product line of different products, rendering testing and other forms of analysis very difficult, since they need to be repeated for every product configuration.

A precise and efficient way of integrating evolving software components is a must to ensure backward compatibilities of component changes with the overall system. In addition, to facilitate modular development and quality assurance, the problem of specifying and verifying inter-component requirements and guarantees becomes paramount.

We plan to study the problem of integrating evolving software components and verifying their compatibility when a change happens in a certain component. To define compatibility, we formalize the notion of *client-specific equivalence*, which is similar to the *property-specific equivalence* used in differential assertion checking [LMSH13] and incremental upgrade checking [SFS12]. But instead of explicitly defining properties as relative specifications or change contracts, we intend to generate equivalence specifications automatically from a static analysis of the source code. Furthermore, if the change made to either component causes incompatibilities, i.e., the compatibility specification is not satisfied, we would like to locate the root cause of the issues and suggest possible fixes.

From Analysis to Construction. For almost any desirable functionality, developers now can find existing code snippets from Q/A websites such as stack overflow, from documentation of third party libraries, or from other open-source projects. Instead of developing the desired functionality from scratch, a more common practice is to copy existing code snippets and appropriately integrate them with the host code base under development. This manual development process includes wiring up correct data flow paths between the copied code and the host code, and inserting necessary glue code at the beginning and the end of the copied code to initialize and clean up relevant data objects. A technique that automates even parts of this development process will be extremely valuable.

This dissertation already demonstrates the potential of understanding and reusing historical change data, and with the increasing volume of code fragments available on the Internet, we plan to look at the problem of *modular software construction* through mining and composition. We are specifically interested in synthesizing new functionality from existing code which may not have been designed for composition a priori. The final goal is to produce abstractions and analyses enabling correct and modular construction of software based on all available code sources.

To achieve the goal, we need to design algorithms to (1) identify reusable software components from multiple sources, (2) establish properties of individual components in the absence of the rest of the system, (3) detect and resolve "semantic" software integration problems related to conflicting program behaviors, and (4) validate composed final products and provide quality assurance through testing, debugging and verification.

The success of this approach will greatly improve not only the productivity of developers, but also the quality of software they produce. It will bring a revolutionary change to the software development process which is powered by massive reuse, in-depth analysis and validation.

Bibliography

[AA14] Pragya Agarwal and Arun Prakash Agrawal. Fault-Localization Techniques for Software Systems: A Literature Review. *ACM SIGSOFT Software Engineering Notes*, 39(5):1–8, September 2014.

[ACSW12] Nele Andersen, Krzysztof Czarnecki, Steven She, and Andrzej Wąsowski. Efficient Synthesis of Feature Models. In *Proceedings of the 16th International Software Product Line Conference - Volume 1*, pages 106–115, New York, NY, USA, 2012. ACM.

[Act17] Apache ActiveMQ. http://activemq.apache.org, 2017.

[AHS$^+$14] Ra'Fat Al-Msie'deen, Marianne Huchard, Abdelhak-Djamel Seriai, Christelle Urtado, Sylvain Vauttier, and Ahmad Al-Khlifat. Concept Lattices: A Representation Space to Structure Software Variability. In *Proceedings of the 5th International Conference on Information and Communication Systems*, pages 1–6, 2014.

[AJB$^+$14] Michał Antkiewicz, Wenbin Ji, Thorsten Berger, Krzysztof Czarnecki, Thomas Schmorleiz, Ralf Lämmel, Stefan Stănciulescu, Andrzej Wąsowski, and Ina Schaefer. Flexible Product Line Engineering with a Virtual Platform. In *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 532–535, New York, NY, USA, 2014. ACM.

[ALHL$^+$15] Wesley K.G. Assunção, Roberto E. Lopez-Herrejon, Lukas Linsbauer, Silvia R. Vergilio, and Alexander Egyed. Extracting Variability-Safe Feature Models from Source Code Dependencies in System Variants. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, pages 1303–1310, New York, NY, USA, 2015. ACM.

[AR11] Mithun Acharya and Brian Robinson. Practical Change Impact Analysis Based on Static Program Slicing for Industrial Software Systems. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 746–755, New York, NY, USA, May 2011. ACM.

[Arn96] Robert S. Arnold. *Software Change Impact Analysis*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1996.

[ASH$^+$13a] Ra'Fat Al-Msie'deen, Abdelhak-Djamel Seriai, Marianne Huchard, Christelle Urtado, and Sylvain Vauttier. Mining Features from the Object-oriented Source Code of Software Variants by Combining Lexical and Structural Similarity. In *Proceedings of the 2013 IEEE 14th International Conference on Information Reuse & Integration*, pages 586–593, 2013.

[ASH+13b] Ra'Fat Al-Msie'deen, Abdelhak-Djamel Seriai, Marianne Huchard, Christelle Urtado, Sylvain Vauttier, and Hamzeh Eyal Salman. Feature Location in a Collection of Software Product Variants Using Formal Concept Analysis. In *Proceedings of the 13th International Conference on Software Reuse*, volume 7925, pages 302–307, 2013.

[Azu16] Microsoft Azure: Ways to Contribute. https://azure.github.io/guidelines, December 2016.

[Bat05] Don Batory. Feature Models, Grammars, and Propositional Formulas. In *Proceedings of the 9th International Conference on Software Product Lines*, pages 7–20, Berlin, Heidelberg, 2005. Springer-Verlag.

[Baz17] Bazel: a Fast, Scalable, Muti-Language, and Extensible Build System. https://bazel.build, 2017.

[BCC+05] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An Overview of JML Tools and Applications. *International Journal on Software Tools for Technology Transfer*, 7(3):212–232, June 2005.

[BCE15] Apache Commons Byte Code Engineering Library. https://commons.apache.org/proper/commons-bcel, 2015.

[BE04] Yuriy Brun and Michael D. Ernst. Finding Latent Code Errors via Machine Learning over Program Executions. In *Proceedings of the 26th International Conference on Software Engineering*, pages 480–490, Washington, DC, USA, 2004. IEEE Computer Society.

[BGH+15] David Binkley, Nicolas Gold, Mark Harman, Syed Islam, Jens Krinke, and Shin Yoo. ORBS and the Limits of Static Slicing. In *Proceedings of the 15th International Working Conference on Source Code Analysis and Manipulation*, pages 1–10, September 2015.

[BHEN13] Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin. Early Detection of Collaboration Conflicts and Risks. *IEEE Transactions on Software Engineering*, 39(10):1358–1375, October 2013.

[Bil05] Philip Bille. A Survey on Tree Edit Distance and Related Problems. *Theoretical Computer Science*, 337(1-3):217–239, June 2005.

[Bit16] How to Contribute Code to Bitcoin Core. https://bitcoincore.org/en/faq/contributing-code, December 2016.

[CCW+01] Annie Chen, Eric Chou, Joshua Wong, Andrew Y. Yao, Qing Zhang, Shao Zhang, and Amir Michail. CVSSearch: Searching through Source Code using CVS Comments. In *Proceedings of the International Conference on Software Maintenance*, 2001.

[CFSM16] Catarina Costa, Jair Figueiredo, Anita Sarma, and Leonardo Murta. TIPMerge: Recommending Developers for Merging Branches. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 998–1002, New York, NY, USA, 2016. ACM.

[CN01] Paul C. Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley Professional, 2001.

[Col16] Apache Commons Collections Library. https://commons.apache.org/proper/commons-collections, December 2016.

[Cow16] Cow: Semantic Version Control. http://jelv.is/cow, 2016.

[CP14] Cristian Cadar and Hristina Palikareva. Shadow Symbolic Execution for Better Testing of Evolving Software. In *Proceedings of the 36th International Conference on Software Engineering*, pages 432–435, New York, NY, USA, 2014. ACM.

[CR00] Kunrong Chen and Václav Rajlich. Case Study of Feature Location Using Dependence Graph. In *Proceedings of the 8th International Workshop on Program Comprehension*, pages 241–247, 2000.

[CRGW96] Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. Change Detection in Hierarchically Structured Information. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 493–504, 1996.

[CSV17] Using Apache Commons CSV. https://commons.apache.org/proper/commons-csv, 2017.

[CW07] Krzysztof Czarnecki and Andrzej Wąsowski. Feature Diagrams and Logics: There and Back Again. In *Proceedings of the 11th International Software Product Line Conference*, pages 23–34, Washington, DC, USA, 2007. IEEE Computer Society.

[Dar16] Understanding Darcs/Patch Theory. http://en.wikibooks.org/wiki/Understanding_Darcs/Patch_theory, March 2016.

[Dat17] DataTables: Table Plug-In for JQuery. https://datatables.net, 2017.

[DCMJ06] Danny Dig, Can Comertoglu, Darko Marinov, and Ralph Johnson. Automated Detection of Refactorings in Evolving Components. In *Proceedings of the 20th European Conference on Object-Oriented Programming*, pages 404–428, Berlin, Heidelberg, 2006. Springer-Verlag.

[DG04] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, pages 137–150, Berkeley, CA, USA, 2004. USENIX Association.

[Dia16] Type Inference for Generic Instance Creation. https://docs.oracle.com/javase/8/docs/technotes/guides/language/type-inference-generic-instance-creation.html, 2016.

[DKW08] Vijay D'Silva, Daniel Kroening, and George Weissenbacher. A Survey of Automated Techniques for Formal Software Verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(7):1165–1178, July 2008.

[DLE03] Nii Dodoo, Lee Lin, and Michael D. Ernst. Selecting, Refining, and Evaluating Predicates for Program Analysis. Technical Report MIT-LCS-TR-914, MIT Laboratory for Computer Science, Cambridge, MA, July 2003.

[DMJN08] Danny Dig, Kashif Manzoor, Ralph E. Johnson, and Tien N. Nguyen. Effective Software Merging in the Presence of Object-Oriented Refactorings. *IEEE Transactions on Software Engineering*, 34(3):321–335, May 2008.

[DRB$^+$13] Yael Dubinsky, Julia Rubin, Thorsten Berger, Slawomir Duszynski, Martin Becker, and Krzysztof Czarnecki. An Exploratory Study of Cloning in Industrial Software Product Lines. In *Proceedings of the 2013 17th European Conference on Software Maintenance and Reengineering*, pages 25–34, 2013.

[DRGP13] Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. Feature Location in Source Code: A Taxonomy and Survey. *Journal of Software: Evolution and Process*, 25(1):53–95, 2013.

[DRP13] Bogdan Dit, Meghan Revelle, and Denys Poshyvanyk. Integrating Information Retrieval, Execution and Link Analysis Algorithms to Improve Feature Location in Software. *Empirical Software Engineering*, 18(2):277–309, Apr 2013.

[ECGN99] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically Discovering Likely Program Invariants to Support Program Evolution. In *Proceedings of the 21st International Conference on Software Engineering*, pages 213–224, New York, NY, USA, 1999. ACM.

[Ela15] Elasticsearch: Distributed, Open Source Search and Analytics Engine. `https://www.elastic.co/products/elasticsearch`, 2015.

[EPG$^+$07] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon System for Dynamic Detection of Likely Invariants. *Science of Computer Programming*, 69(1-3):35–45, December 2007.

[ERS10] Emelie Engström, Per Runeson, and Mats Skoglund. A Systematic Review on Regression Test Selection Techniques. *Information and Software Technology*, 52(1):14–30, January 2010.

[FAW09] Javed Ferzund, Syed Nadeem Ahsan, and Franz Wotawa. Empirical Evaluation of Hunk Metrics as Bug Predictors. In *Proceedings of the International Conferences on Software Process and Product Measurement*, pages 242–254, Berlin, Heidelberg, 2009. Springer-Verlag.

[FG06] Beat Fluri and Harald C. Gall. Classifying Change Types for Qualifying Change Couplings. In *Proceedings of the 14th IEEE International Conference on Program Comprehension*, pages 35–45. IEEE, 2006.

[FGS14] Grigory Fedyukovich, Arie Gurfinkel, and Natasha Sharygina. Incremental Verification of Compiler Optimizations. In *Proceedings of the 6th International Symposium on NASA Formal Methods*, volume 8430, pages 300–306, New York, NY, USA, 2014. Springer-Verlag New York, Inc.

[FMB$^+$14] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. Fine-grained and Accurate Source Code Differencing. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, pages 313–324, September 2014.

[Fow99] Martin Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, Boston, MA, USA, 1999.

[FRC81] Kurt F. Fischer, Farzad Raji, and Andrew Chruscicki. A Methodology for Retesting Modified Software. In *Proceedings of the National Telecommunications Conference*, pages 1–6, November 1981.

[FWPG07] Beat Fluri, Michael Wuersch, Martin Pinzger, and Harald Gall. Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction. *IEEE Transactions on Software Engineering*, 33(11):725–743, November 2007.

[GEM15] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. Practical Regression Test Selection with Dynamic File Dependencies. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 211–222, New York, NY, USA, 2015. ACM.

[GHK$^+$01] Todd L. Graves, Mary Jean Harrold, Jung-Min Kim, Adam Porter, and Gregg Rothermel. An Empirical Study of Regression Test Selection Techniques. *ACM Transactions on Software Engineering and Methodology*, 10(2):184–208, April 2001.

[Git16a] Git: git-bisect Documentation. http://git-scm.com/docs/git-bisect, 2016.

[Git16b] Git Version Control System. https://git-scm.com, 2016.

[Git16c] Git: Submitting Patches. https://github.com/git/git/blob/master/Documentation/SubmittingPatches, 2016.

[Git17a] GitFlowVisualize. https://www.npmjs.com/package/git-flow-vis, 2017.

[Git17b] GitHub: The World's Leading Software Development Platform. https://github.com, 2017.

[GLRG11] Patrice Godefroid, Shuvendu K. Lahiri, and Cindy Rubio-González. Statically Validating Must Summaries for Incremental Compositional Dynamic Test Generation. In *Proceedings of the 18th International Conference on Static Analysis*, pages 112–128, Berlin, Heidelberg, 2011. Springer-Verlag.

[GMS$^+$14] Milos Gligoric, Rupak Majumdar, Rohan Sharma, Lamyaa Eloussi, and Darko Marinov. Regression Test Selection for Distributed Software Histories. In *Proceedings of the 16th International Conference on Computer Aided Verification*, volume 8559, pages 293–309, New York, NY, USA, July 2014. Springer International Publishing.

[God07] Patrice Godefroid. Compositional Dynamic Test Generation. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 47–54, New York, NY, USA, 2007. ACM.

[GW05] Carsten Görg and Peter Weißgerber. Detecting and Visualizing Refactorings from Software Archives. In *Proceedings of the 13th International Workshop on Program Comprehension*, 2005.

[Had15] Apache Hadoop Project. https://hadoop.apache.org, 2015.

[Har12] Dick Hardt. The OAuth 2.0 Authorization Framework. RFC 6749, RFC Editor, Fremont, CA, USA, October 2012.

[HL02] Sudheendra Hangal and Monica S. Lam. Tracking Down Software Bugs Using Automatic Anomaly Detection. In *Proceedings of the 24th International Conference on Software Engineering*, pages 291–301, New York, NY, USA, 2002. ACM.

[HLHE11] Evelyn Nicole Haslinger, Roberto E. Lopez-Herrejon, and Alexander Egyed. Reverse Engineering Feature Models from Programs' Feature Sets. In *Proceedings of the 18th Working Conference on Reverse Engineering*, pages 308–312, 2011.

[HM08] Masatomo Hashimoto and Akira Mori. Diff/TS: A Tool for Fine-Grained Structural Change Analysis. In *Proceedings of the 15th Working Conference on Reverse Engineering*, pages 279–288, Antwerp, October 2008. IEEE.

[HOZ+12] Shinpei Hayashi, Takayuki Omori, Teruyoshi Zenmyo, Katsuhisa Maruyama, and Motoshi Saeki. Refactoring Edit History of Source Code. In *Proceedings of the 28th IEEE International Conference on Software Maintenance*, pages 617–620. IEEE, September 2012.

[HZ13] Kim Herzig and Andreas Zeller. The Impact of Tangled Code Changes. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 121–130, Piscataway, NJ, USA, 2013. IEEE Press.

[IO16] Commons IO: Apache Commons IO Library. https://commons.apache.org/proper/commons-io, December 2016.

[IPW01] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: A Minimal Core Calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, May 2001.

[Jac16] JaCoCo Java Code Coverage Library. http://www.eclemma.org/jacoco, 2016.

[JGi16] JGit: A Lightweight, Pure Java Library Implementing the Git Version Control System. https://eclipse.org/jgit, 2016.

[JIR17] JIRA Software. https://www.atlassian.com/software/jira, 2017.

[JOX10] Wei Jin, Alessandro Orso, and Tao Xie. Automated Behavioral Regression Testing. In *Proceedings of the 2010 3rd International Conference on Software Testing, Verification and Validation*, pages 137–146, Washington, DC, USA, 2010. IEEE Computer Society.

[JSO17] Introducing JSON. http://www.json.org, 2017.

[JST17] jsTree: JQuery Tree Plugin. https://www.jstree.com, 2017.

[JUn16] A Unit Testing Framework for the Java Programming Language. http://junit.org, 2016.

[KA08] Christian Kästner and Sven Apel. Type-Checking Software Product Lines - a Formal Approach. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 258–267, Washington, DC, USA, 2008. IEEE Computer Society.

[Kah62] Arthur B. Kahn. Topological Sorting of Large Networks. *Communications of the ACM*, 5(11):558–562, November 1962.

[KAT+09] Christian Kästner, Sven Apel, Salvador Trujillo, Martin Kuhlemann, and Don Batory. Guaranteeing Syntactic Correctness for all Product Line Variants: A Language-Independent Approach. In *Proceedings of the 47th International Conference on Objects, Models, Components, Patterns (TOOLS EUROPE)*, volume 33, pages 175–194, Zurich, Switzerland, June 2009. Springer Berlin Heidelberg.

[Kin76] James C. King. Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7):385–394, July 1976.

[KKL+98] Kyo C. Kang, Sajoong Kim, Jaejoon Lee, Kijoo Kim, Euiseob Shin, and Moonhang Huh. FORM: A Feature-oriented Reuse Method with Domain-specific Reference Architectures. *Annals of Software Engineering*, 5(1):143–168, 1998.

[KP02] Jung-Min Kim and Adam Porter. A History-based Test Prioritization Technique for Regression Testing in Resource Constrained Environments. In *Proceedings of the 24th International Conference on Software Engineering*, pages 119–129, New York, NY, USA, 2002. ACM.

[KR11] David Kawrykow and Martin P. Robillard. Non-essential Changes in Version Histories. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 351–360, New York, NY, USA, 2011. ACM.

[KWZ08] Sunghun Kim, E. James Whitehead, Jr., and Yi Zhang. Classifying Software Changes: Clean or Buggy? *IEEE Transactions on Software Engineering*, 34(2):181–196, March 2008.

[LAG+14] Lukas Linsbauer, Florian Angerer, Paul Grünbacher, Daniela Lettner, Herbert Prähofer, Roberto E. Lopez-Herrejon, and Alexander Egyed. Recovering Feature-To-Code Mappings in Mixed-Variability Software Systems. In *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution*, pages 426–430, 2014.

[Lin17] Submitting Patches: the Essential Guide to Getting Your Code into the Kernel. https://www.kernel.org/doc/Documentation/process/submitting-patches.rst, 2017.

[LLE13] Lukas Linsbauer, Roberto E. Lopez-Herrejon, and Alexander Egyed. Recovering Traceability Between Features and Code in Product Variants. In *Proceedings of the 17th International Software Product Line Conference*, pages 131–140, 2013.

[LLHE16] Lukas Linsbauer, Roberto Erick Lopez-Herrejon, and Alexander Egyed. Variability Extraction and Modeling for Product Variants. *Software & Systems Modeling*, pages 1–21, 2016.

[LMPR07] Dapeng Liu, Andrian Marcus, Denys Poshyvanyk, and Vaclav Rajlich. Feature Location via Information Retrieval Based Filtering of a Single Scenario Execution Trace. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, pages 234–243, New York, NY, USA, 2007. ACM.

[LMSH13] Shuvendu K. Lahiri, Kenneth L. McMillan, Rahul Sharma, and Chris Hawblitzel. Differential Assertion Checking. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 345–355, New York, NY, USA, 2013. ACM.

[LR03] James Law and Gregg Rothermel. Whole Program Path-Based Dynamic Impact Analysis. In *Proceedings of the 25th International Conference on Software Engineering*, pages 308–318. IEEE, May 2003.

[LRC15] Yi Li, Julia Rubin, and Marsha Chechik. Semantic Slicing of Software Version Histories. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, pages 686–696, Lincoln, NE, USA, November 2015.

[LVH10] Shuvendu K Lahiri, Kapil Vaswani, and C AR Hoare. Differential Static Analysis: Opportunities, Applications, and Challenges. In *Proceedings of the 2010 FSE/SDP Workshop on the Future of Software Engineering Research*, pages 201–204, New York, NY, USA, 2010. ACM.

[LZRC16] Yi Li, Chenguang Zhu, Julia Rubin, and Marsha Chechik. Precise Semantic History Slicing through Dynamic Delta Refinement. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 495–506, Singapore, Singapore, September 2016.

[LZRC17a] Yi Li, Chenguang Zhu, Julia Rubin, and Marsha Chechik. FHistorian: Locating Features in Version Histories. In *Proceedings of the 21st International Systems and Software Product Line Conference - Volume A*, pages 49–58, New York, NY, USA, 2017. ACM.

[LZRC17b] Yi Li, Chenguang Zhu, Julia Rubin, and Marsha Chechik. Semantic Slicing of Software Version Histories. *IEEE Transactions on Software Engineering*, 44(2):182–201, Feburary 2017.

[LZRC18] Yi Li, Chenguang Zhu, Julia Rubin, and Marsha Chechik. CSlicerCloud: A Web-Based Semantic History Slicing Framework. In *Proceedings of the 40th International Conference on Software Engineering*, 2018.

[Mat16] Commons Math: The Apache Commons Mathematics Library. https://commons.apache.org/proper/commons-math, December 2016.

[Mer16] Mercurial Source Control Management System. http://mercurial.selenic.com, 2016.

[MHPB12] Emerson Murphy-Hill, Chris Parnin, and Andrew P. Black. How We Refactor, and How We Know It. *IEEE Transactions on Software Engineering*, 38(1):5–18, Jan 2012.

[Mol09] Ian Molyneaux. *The Art of Application Performance Testing: Help for Programmers and Quality Assurance*. O'Reilly Media, Inc., 1st edition, 2009.

[MS06] Ghassan Misherghi and Zhendong Su. HDD: Hierarchical Delta Debugging. In *Proceedings of the 28th International Conference on Software Engineering*, pages 142–151, New York, NY, USA, 2006. ACM.

[MSBE15] Kivanç Muşlu, Luke Swart, Yuriy Brun, and Michael D. Ernst. Development History Granularity Transformations. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, pages 697–702, Lincoln, NE, USA, November 2015.

[Mvn15] Apache Maven Project. https://maven.apache.org, 2015.

[Mvn16] Guide to Developing Maven. http://maven.apache.org/guides/development/guide-maven-development.html, 2016.

[NBKC14] Sarah Nadi, Thorsten Berger, Christian Kästner, and Krzysztof Czarnecki. Mining Configuration Constraints: Static Analyses and Empirical Results. In *Proceedings of the 36th International Conference on Software Engineering*, pages 140–151, New York, NY, USA, 2014. ACM.

[NHC$^+$16] Anh Tuan Nguyen, Michael Hilton, Mihai Codoban, Hoan Anh Nguyen, Lily Mast, Eli Rademacher, Tien N. Nguyen, and Danny Dig. API Code Recommendation Using Statistical Learning from Fine-grained Changes. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 511–522, New York, NY, USA, 2016. ACM.

[Nod17] Node.js. https://nodejs.org, September 2017.

[OAH03] Alessandro Orso, Taweesup Apiwattanapong, and Mary Jean Harrold. Leveraging Field Data for Impact Analysis and Regression Testing. In *Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 128–137, New York, NY, USA, 2003. ACM.

[OAL$^+$04] Alessandro Orso, Taweesup Apiwattanapong, James Law, Gregg Rothermel, and Mary Jean Harrold. An Empirical Comparison of Dynamic Impact Analysis Algorithms. In *Proceedings of the 26th International Conference on Software Engineering*, pages 491–500, Washington, DC, USA, May 2004. IEEE Computer Society.

[OX08] Alessandro Orso and Tao Xie. BERT: BEhavioral Regression Testing. In *Proceedings of the 2008 International Workshop on Dynamic Analysis: Held in Conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 36–42, New York, NY, USA, July 2008.

[PBL05] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[PDHP08] Suzette Person, Matthew B Dwyer, Avery Hall, and Corina Păsăreanu. Differential Symbolic Execution. In *Proceedings of the 16th International Symposium on the Foundations of Software Engineering*, pages 226–237, Atlanta, GA, USA, March 2008.

[PE04] Jeff H. Perkins and Michael D. Ernst. Efficient Incremental Algorithms for Dynamic Detection of Likely Invariants. In *Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering*, pages 23–32, New York, NY, USA, 2004. ACM.

[PL16] Rachel Potvin and Josh Levenberg. Why Google Stores Billions of Lines of Code in a Single Repository. *Communications of the ACM*, 59(7):78–87, June 2016.

[PMH+14] Fabrizio Pastore, Leonardo Mariani, Antti E. J. Hyvärinen, Grigory Fedyukovich, Natasha Sharygina, Stephan Sehestedt, and Ali Muhammad. Verification-Aided Regression Testing. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 37–48, New York, NY, USA, 2014. ACM.

[PYRK11] Suzette Person, Guowei Yang, Neha Rungta, and Sarfraz Khurshid. Directed Incremental Symbolic Execution. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 504–515, New York, NY, USA, 2011. ACM.

[QYR12] Dawei Qi, Jooyong Yi, and Abhik Roychoudhury. Software Change Contracts. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, pages 22:1–22:4, New York, NY, USA, 2012. ACM.

[RC13a] Julia Rubin and Marsha Chechik. A Framework for Managing Cloned Product Variants. In *Proceedings of the 35th International Conference on Software Engineering*, pages 1233–1236, 2013.

[RC13b] Julia Rubin and Marsha Chechik. A Survey of Feature Location Techniques. In Iris Reinhartz-Berger et al., editor, *Domain Engineering: Product Lines, Conceptual Models, and Languages.*, pages 29–58. Springer, 2013.

[RCC13] Julia Rubin, Krzysztof Czarnecki, and Marsha Chechik. Managing Cloned Variants: a Framework and Experience. In *Proceedings of the 17th International Software Product Line Conference*, pages 101–110, 2013.

[RCC15] Julia Rubin, Krzysztof Czarnecki, and Marsha Chechik. Cloned Product Variants: From Ad-Hoc to Managed Software Product Lines. *Journal on Software Tools for Technology Transfer*, 17(5):627–646, 2015.

[RDP10] Meghan Revelle, Bogdan Dit, and Denys Poshyvanyk. Using Data Fusion and Web Mining to Support Feature Location in Software. In *Proceedings of the 2010 IEEE 18th International Conference on Program Comprehension*, pages 14–23, Washington, DC, USA, 2010. IEEE Computer Society.

[RH94] Gregg Rothermel and Mary Jean Harrold. A Framework for Evaluating Regression Test Selection Techniques. In *Proceedings of the 16th International Conference on Software Engineering*, pages 201–210, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[RH96] Gregg Rothermel and Mary Jean Harrold. Analyzing Regression Test Selection Techniques. *IEEE Transactions on Software Engineering*, 22(8):529–551, Aug 1996.

[RKBC12] Julia Rubin, Andrei Kirshin, Goetz Botterweck, and Marsha Chechik. Managing Forked Product Variants. In *Proceedings of the 16th International Software Product Line Conference*, volume 1, pages 156–160, New York, NY, USA, 2012. ACM.

[RPK11] Uwe Ryssel, Joern Ploennigs, and Klaus Kabitzsch. Extraction of Feature Models from Formal Contexts. In *Proceedings of the 15th International Software Product Line Conference, Volume 2*, pages 4:1–4:8, New York, NY, USA, 2011. ACM.

[RST+04] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G. Ryder, and Ophelia Chesley. Chianti: A Tool for Change Impact Analysis of Java Programs. In *Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 432–448, New York, NY, USA, 2004. ACM.

[Sem16] The Diff and Merge Tool that Understands Your Code – SemanticMerge. https://www.semanticmerge.com, 2016.

[SFS12] Ondrej Sery, Grigory Fedyukovich, and Natasha Sharygina. Incremental Upgrade Checking by Means of Interpolation-Based Function Summaries. In *Proceedings of the 2012 Formal Methods in Computer-Aided Design*, pages 114–121, Oct 2012.

[SG17] Ripon Saha and Milos Gligoric. Selective Bisection Debugging. In *Proceedings of the 20th International Conference on Fundamental Approaches to Software Engineering*, pages 60–77, New York, NY, USA, 2017. Springer-Verlag New York, Inc.

[SJ12] Francisco Servant and James A. Jones. WhoseFault: Automatic Developer-To-Fault Assignment Through Fault Localization. In *Proceedings of the 34th International Conference on Software Engineering*, pages 36–46, Piscataway, NJ, USA, 2012. IEEE Press.

[SLB+11] Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wąsowski, and Krzysztof Czarnecki. Reverse Engineering Feature Models. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 461–470, New York, NY, USA, 2011. ACM.

[SLKP13] Ripon K. Saha, Matthew Lease, Sarfraz Khurshid, and Dewayne E. Perry. Improving Bug Localization Using Structured Information Retrieval. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, pages 345–355, Piscataway, NJ, USA, 2013. IEEE Press.

[SLL+15] Xiaobing Sun, Bixin Li, Hareton Leung, Bin Li, and Junwu Zhu. Static Change Impact Analysis Techniques. *Journal of Systems and Software*, 109(C):137–149, November 2015.

[Som04] Ian Sommerville. *Software Engineering (7th Edition)*. Pearson Addison Wesley, 2004.

[SSS14] Anas Shatnawi, Abdelhak Seriai, and Houari Sahraoui. Recovering Architectural Variability of a Family of Product Variants. In *Proceedings of the 14th International Conference on Software Reuse*, pages 17–33, 2014.

[Sur17] Maven Surefire Plugin. http://maven.apache.org/surefire/maven-surefire-plugin, 2017.

[SVN16] Apache Subversion (SVN) Version Control System. http://subversion.apache.org, 2016.

[SZ11] David Schuler and Andreas Zeller. Assessing Oracle Quality with Checked Coverage. In *Proceedings of the 4th IEEE International Conference on Software Testing, Verification and Validation*, pages 90–99, Washington, DC, USA, 2011. IEEE Computer Society.

[Tip95] Frank Tip. A Survey of Program Slicing Techniques. *Journal of Programming Languages*, 3:121–189, 1995.

[TKW+13] Ryosuke Tsuchiya, Tadahisa Kato, Hironori Washizaki, Masumi Kawakami, Yoshiaki Fukazawa, and Kentaro Yoshimura. Recovering Traceability Links between Requirements and Source Code in the Same Series of Software Products. In *Proceedings of the 17th International Software Product Line Conference*, pages 121–130, New York, NY, USA, 2013.

[Try16] The try-with-resources Statement. https://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html, March 2016.

[VRCG+99] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a Java Bytecode Optimization Framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, pages 125–135. IBM Press, 1999.

[WC10] Byron J. Williams and Jeffrey C. Carver. Characterizing Software Architecture Changes: A Systematic Review. *Information and Software Technology*, 52(1):31–51, January 2010.

[WE03] Joel Winstead and David Evans. Towards Differential Program Analysis. In *Proceedings of the ICSE Workshop on Dynamic Analysis*, pages 37–40, 2003.

[Wei81] Mark Weiser. Program Slicing. In *Proceedings of the 5th International Conference on Software Engineering*, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[WH05] Chadd C. Williams and Jeffrey K. Hollingsworth. Automatic Mining of Source Code Repositories to Improve Bug Finding Techniques. *IEEE Transactions on Software Engineering*, 31(6):466–480, June 2005.

[WL16] Shaowei Wang and David Lo. AmaLgam+: Composing Rich Information Sources for Accurate Bug Localization. *Journal of Software: Evolution and Process*, 28(10):921–942, 2016.

[WS95] Norman Wilde and Michael C. Scully. Software Reconnaissance: Mapping Program Features to Code. *Journal of Software Maintenance*, 7:49–62, 1995.

[XN05] Tao Xie and David Notkin. Checking inside the Black Box: Regression Testing by Comparing Value Spectra. *IEEE Transactions on Software Engineering*, 31(10):869–883, October 2005.

[XXJ12] Yinxing Xue, Zhenchang Xing, and Stan Jarzabek. Feature Location in a Collection of Product Variants. In *Proceedings of the 2012 19th Working Conference on Reverse Engineering*, pages 145–154, Washington, DC, USA, 2012. IEEE Computer Society.

[YAM17] YAML Ain't Markup Language. http://www.yaml.org/, 2017.

[YH12] Shin Yoo and Mark Harman. Regression Testing Minimization, Selection and Prioritization: A Survey. *Software Testing, Verification & Reliability*, 22(2):67–120, March 2012.

[YKPR14] Guowei Yang, Sarfraz Khurshid, Suzette Person, and Neha Rungta. Property Differencing for Incremental Checking. In *Proceedings of the 36th International Conference on Software Engineering*, pages 1059–1070, New York, NY, USA, 2014. ACM.

[YQTR15] Jooyong Yi, Dawei Qi, Shin Hwei Tan, and Abhik Roychoudhury. Software Change Contracts. *ACM Transactions on Software Engineering and Methodology*, 24(3):18:1–18:43, May 2015.

[ZDZ03] Thomas Zimmermann, Stephan Diehl, and Andreas Zeller. How History Justifies System Architecture (or Not). In *Proceedings of the 6th International Workshop on Principles of Software Evolution*, pages 73–83, Washington, DC, USA, 2003. IEEE Computer Society.

[Zel99] Andreas Zeller. Yesterday, My Program Worked. Today, It Does Not. Why? In *Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 253–267, London, UK, UK, 1999. Springer-Verlag.

[ZFdSZ12] Tewfik Ziadi, Luz Frias, Marcos Aurélio Almeida da Silva, and Mikal Ziane. Feature Identification from the Source Code of Product Variants. In *2012 16th European Conference on Software Maintenance and Reengineering*, pages 417–422, March 2012.

[ZH02] Andreas Zeller and Ralf Hildebrandt. Simplifying and Isolating Failure-Inducing Input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002.

[ZKK11] Lingming Zhang, Miryung Kim, and Sarfraz Khurshid. Localizing Failure-Inducing Program Edits Based on Spectrum Information. In *Proceedings of the 27th International Conference on Software Maintenance*, pages 23–32. IEEE, 2011.

[ZLRC17] Chenguang Zhu, Yi Li, Julia Rubin, and Marsha Chechik. A Dataset for Dynamic Discovery of Semantic Changes in Version Controlled Software Histories. In *Proceedings of the 14th International Conference on Mining Software Repositories*, pages 523–526, Piscataway, NJ, USA, 2017. IEEE Press.

[ZLX$^+$16] Yun Zhang, David Lo, Xin Xia, Tien-Duy B. Le, Giuseppe Scanniello, and Jianling Sun. Inferring Links between Concerns and Methods with Multi-Abstraction Vector Space Model. In *Proceedings of the 32nd IEEE International Conference on Software Maintenance and Evolution*, pages 110–121, Oct 2016.

[ZWDZ04] Thomas Zimmermann, Peter Weißgerber, Stephan Diehl, and Andreas Zeller. Mining Version Histories to Guide Software Changes. In *Proceedings of the 26th International Conference on Software Engineering*, pages 563–572, Washington, DC, USA, 2004. IEEE Computer Society.

Appendix A

A Dataset of Semantic Changes in Version Histories

A.1 Dataset Descriptions

Here we present a dataset for dynamic discovery of semantic changes in version controlled software histories. The dataset contains 81 semantic change data items collected from eight open source Java projects found on GitHub, which covers a broad range of software projects and a significant number of data items. The dataset can be accessed from: `https://github.com/Chenguang-Zhu/DoSC`.

Each data item in the dataset is associated with a meta-data file written in the YAML [YAM17] format. Figure A.1 shows an example of such a file for the functionality `CALCITE-1168`.[1] In each meta-data file:

- `id` is the issue key of the functionality on JIRA – a unique identifier originally assigned by developers in the issue tracking system.
- `description` is the developers' description of the functionality, found on the JIRA release notes page.
- `project` designates the project in which the functionality belongs. The project's `name` and `url` are provided.
- `issue url` designates the link of the issue report of the functionality on JIRA. The issue report page contains detailed information and activity log of the functionality.
- `history start` specifies the starting point of the history segment where the functionality was developed. It is the SHA-1 ID of a release commit, which is the closest release version before the functionality was developed.
- `history end` specifies the ending point of the history segment where the functionality was developed. It is the SHA-1 ID of the closest release version after the functionality was developed.
- `test suite` designates the associated test suite of the functionality. The test suite exercises the behaviors of the functionality. All the test methods of the test suite are listed in this field.

[1]`https://issues.apache.org/jira/browse/CALCITE-1168`

- `history slice` designates the 1-minimal semantic history slice with respect to the functionality, i.e., the ground truth for semantic history slicing.

```
id: "CALCITE-1168"
description: "Add DESCRIBE SCHEMA/DATABASE/TABLE/query"
project:
  name: "Calcite"
  url: "https://github.com/apache/calcite"
issue url: "https://issues.apache.org/jira/browse/CALCITE-1168"
history start: "8eebfc6d"
history end: "aeb6bf14"
test suite:
  - "SqlParserTest.testDescribeSchema"
  - "SqlParserTest.testDescribeTable"
  - "SqlParserTest.testDescribeStatement"
history slice:
  - "a065200a"
  - "da875a67"
```

Figure A.1: Meta-data file of the functionality `CALCITE-1168`.

Table A.1 presents detailed descriptions of the dataset. Each row in the table represents a target functionality. The first eight columns list statistics of the selected functionalities with corresponding history segments and test suites, while the last two columns contain information of the ground truth for semantic history slicing and feature location.

- Column "Functionality ID" designates the JIRA issue key of the functionality - a unique identifier assigned to the functionality by developers in the JIRA issue tracking system.
- Column "H.Start" shows the starting point of the history segment where the functionality was developed. It is the SHA-1 ID of a release commit, which is the last release version before the functionality was developed.
- Column "H.End" shows the ending point of the history segment where the functionality was developed. It is the SHA-1 ID of the earliest release version after the functionality was developed.
- Column "#C" represents the length of history of developing a functionality, measured by the number of commits.
- Column"#F" shows the number of files edited during the development of the functionality.
- Column "#LOC+" designates the number of code lines inserted during the development of the functionality.
- Column "#LOC-" exhibits the number of code lines deleted during the development of the functionality.
- Column "#T" represents the number of test cases in the associated test suite of the functionality.
- Column "Slice" designates the size of the 1-minimal history slice of each functionality, expressed as the number of commits.

- Column "#Dev.Labels" designates the size of the set of commits that developers labeled as part of the functionality.

A.2 Using the Dataset

The dataset can be used for the evaluation of both the semantic history slicing and dynamic feature location techniques.

Semantic History Slicing. The dataset allows researchers to assess the capabilities of their history slicing tools and easily compare with other techniques in the same category. To use the dataset as a benchmark for semantic history slicing, users need to follow the following steps:

1. Pick a functionality that they would like to analyze from the dataset. View its meta-data file.
2. Access the repository of the project via the link provided in the *project url* field of the meta-data.
3. Use the `git clone` command to clone the project repository to the user's local file system.
4. Extract the names of all test methods listed in the *test suite* field of the meta-data.
5. Use the extracted test cases and the history segment specified by the starting point (field `history start`) and the ending point (field `history end`) as the input on which to run the history slicing tool.
6. Compare the resulting semantic history slice with the 1-minimal ground truth we provide (field `history slice`).
7. Repeat the steps 1-6 until the evaluation is completed.

Dynamic Feature Location. In addition to semantic history slicing, the dataset can also be used for other software analysis tasks such as *dynamic feature location* [RC13b], requiring only simple modifications to the above method.

Feature location aims to identify software components that implement a specific program functionality. Dynamic location techniques monitor executions of some target feature and analyze runtime traces collected during the executions, to identify the set of related program entities for the feature. In the dataset, each target functionality is accompanied by a test suite capturing its behaviors and a set of essential commits implementing the functionality. The test suites provided in the dataset can be used to activate and gather execution traces for the corresponding features, while the set of essential commits can be mapped to relevant code entities to evaluate the effectiveness of the feature location techniques.

Table A.1: Detailed statistics of the dataset.

Functionality ID	H.Start	H.End	#C	#F	#LOC+	#LOC-	#T	Slice	#Dev.Labels
LANG-825	bae9f7c3	15a51f1d	475	265	27,630	11,935	2	118	1
LANG-839	bae9f7c3	15a51f1d	475	265	27,630	11,935	2	200	4
LANG-841	bae9f7c3	15a51f1d	475	265	27,630	11,935	2	200	1
LANG-906	bae9f7c3	15a51f1d	475	265	27,630	11,935	5	1	1
LANG-834	c4ecd75	66a3717	179	121	6,889	1,807	12	12	1
LANG-944	c4ecd75	66a3717	179	121	6,889	1,807	1	24	1
LANG-993	24767d6	76cc69c	262	146	6,741	2,076	10	6	1
LANG-999	24767d6	76cc69c	262	146	6,741	2,076	5	15	1
LANG-1006	24767d6	76cc69c	262	146	6,741	2,076	2	14	1
LANG-1033	24767d6	76cc69c	262	146	6,741	2,076	1	22	1
LANG-1088	24767d6	76cc69c	262	146	6,741	2,076	2	1	2
LANG-536	24767d6	76cc69c	262	146	6,741	2,076	17	30	1
LANG-883	24767d6	76cc69c	262	146	6,741	2,076	1	36	1
LANG-1015	24767d6	76cc69c	262	146	6,741	2,076	9	39	1
LANG-1021	24767d6	76cc69c	262	146	6,741	2,076	16	28	1
LANG-1080	24767d6	76cc69c	262	146	6,741	2,076	8	38	1
LANG-1093	24767d6	76cc69c	262	146	6,741	2,076	2	63	2
LANG-1050	0d5d666	36f98d8	515	309	18,885	6,395	4	8	1
LANG-1074	0d5d666	36f98d8	515	309	18,885	6,395	9	6	1
LANG-1119	0d5d666	36f98d8	515	309	18,885	6,395	1	1	3
CALCITE-627	f10ea367	d60f2aa3	51	135	8,274	1,446	2	19	1
CALCITE-655	f10ea367	d60f2aa3	51	135	8,274	1,446	1	19	1
CALCITE-674	d60f2aa3	495f1859	59	196	14,861	9,173	1	11	1
CALCITE-718	495f1859	0c0c203d	92	304	21,348	7,686	1	14	1
CALCITE-758	495f1859	0c0c203d	92	304	21,348	7,686	1	1	2
CALCITE-811	495f1859	0c0c203d	92	304	21,348	7,686	1	1	1
CALCITE-803	495f1859	0c0c203d	92	304	21,348	7,686	1	1	1

CALCITE-925	0c0c203d	ba6e43c6	120	468	68,314	6,096	3	1	1
CALCITE-767	ba6e43c6	c4d346b0	103	465	31,647	13,594	1	8	1
CALCITE-996	ba6e43c6	c4d346b0	103	465	31,647	13,594	1	1	1
CALCITE-1003	ba6e43c6	c4d346b0	103	465	31,647	13,594	25	14	1
CALCITE-1028	ba6e43c6	c4d346b0	103	465	31,647	13,594	1	6	1
CALCITE-1168	8eebfc6d	aeb6bf14	122	399	30,975	4,800	3	2	2
CALCITE-1200	8eebfc6d	aeb6bf14	122	399	30,975	4,800	3	2	1
CALCITE-991	aeb6bf14	08c56b15	78	295	14,908	3,637	5	1	1
CALCITE-1288	aeb6bf14	08c56b15	78	295	14,908	3,637	1	6	1
CALCITE-1309	aeb6bf14	08c56b15	78	295	14,908	3,637	8	7	1
CALCITE-1337	aeb6bf14	08c56b15	78	295	14,908	3,637	2	5	1
MNG-4904	b175144	308d4d4	51	78	1,816	713	1	7	1
MNG-4909	b175144	308d4d4	51	78	1,816	713	2	7	1
MNG-4910	b175144	308d4d4	51	78	1,816	713	1	7	1
MNG-4953	38ced22	0023226	47	96	2,448	329	1	6	1
MNG-5159	089a9f8	6d37598	120	318	3,003	1,098	4	2	1
MNG-5530	b7e3ce2	ea8b2b0	97	160	4,431	4,144	1	1	2
MNG-5549	b7e3ce2	ea8b2b0	97	160	4,431	4,144	1	13	2
MNG-5754	d13c288	cab6659	97	235	9,500	3,930	4	8	3
MNG-5755	d13c288	cab6659	97	235	9,500	3,930	5	7	1
MNG-5767	d13c288	cab6659	97	235	9,500	3,930	3	21	6
MNG-5805	0ddab5f	bb52d85	98	341	3,751	3,030	2	11	4
COMPRESS-295	083e7a4	1dcab3f	169	181	6,638	1,580	2	1	2
COMPRESS-296	083e7a4	1dcab3f	169	181	6,638	1,580	3	37	2
COMPRESS-313	083e7a4	1dcab3f	169	181	6,638	1,580	3	40	1
COMPRESS-327	99bc508	b29395d	148	144	4,644	2,006	18	26	10
COMPRESS-368	99bc508	b29395d	148	144	4,644	2,006	6	12	6
COMPRESS-369	99bc508	b29395d	148	144	4,644	2,006	2	10	2
COMPRESS-373	99bc508	b29395d	148	144	4,644	2,006	1	14	1
COMPRESS-374	99bc508	b29395d	148	144	4,644	2,006	8	15	1

COMPRESS-375	99bc508	b29395d	148	144	4,644	2,006	2	1	2
FLUME-1710	31d45f1b	f7560038	133	258	15,949	2,783	1	1	1
FLUME-2052	cda3bd10	31d45f1b	101	181	14,742	3,097	5	3	1
FLUME-2056	cda3bd10	31d45f1b	101	181	14,742	3,097	1	5	1
FLUME-2130	cda3bd10	31d45f1b	101	181	14,742	3,097	1	3	1
FLUME-2206	cda3bd10	31d45f1b	101	181	14,742	3,097	1	4	1
FLUME-2498	f7560038	5e400ea8	100	428	17,341	8,187	17	65	1
FLUME-2628	f7560038	5e400ea8	100	428	17,341	8,187	7	1	1
FLUME-2955	f7560038	5e400ea8	100	428	17,341	8,187	1	65	1
FLUME-2982	f7560038	5e400ea8	100	428	17,341	8,187	2	35	1
PDFBOX-3307	a281f71	9e102f2	37	42	1,138	268	2	1	4
PDFBOX-3069	3b5ae83	5848e90	272	255	9,737	5,398	2	1	1
PDFBOX-3418	3b5ae83	5848e90	272	255	9,737	5,398	2	3	2
PDFBOX-3461	3b5ae83	5848e90	272	255	9,737	5,398	24	3	1
PDFBOX-3262	7c1a2c8	69a8e03	162	135	3,295	814	1	2	3
CONFIGURATION-466	5270237	f81ff1a	252	694	79,920	80,096	3	13	1
CONFIGURATION-624	89428f1	9fb4ad8	50	34	1,201	655	11	48	7
CONFIGURATION-626	89428f1	9fb4ad8	50	34	1,201	655	4	1	5
NET-436	d8812a3	4c3860e	77	99	2,357	774	5	7	1
NET-525	d483631	abd6711	269	233	6,845	2,393	14	40	2
NET-527	d483631	abd6711	269	233	6,845	2,393	1	40	1
CSV-159	b230a6f5	7310e5c6	79	28	1,640	713	1	10	1
CSV-175	b230a6f5	7310e5c6	79	28	1,640	713	11	48	3
CSV-179	b230a6f5	7310e5c6	79	28	1,640	713	1	56	1
CSV-180	b230a6f5	7310e5c6	79	28	1,640	713	2	56	1
IO-126	61519de4	f6724182	140	140	7,365	1,242	2	6	1
IO-129	61519de4	f6724182	140	140	7,365	1,242	7	10	2
IO-130	61519de4	f6724182	140	140	7,365	1,242	4	11	1
IO-135	61519de4	f6724182	140	140	7,365	1,242	4	23	2
IO-138	61519de4	f6724182	140	140	7,365	1,242	7	13	1

IO-144	61519de4	f6724182	140	140	7,365	1,242	2	1	1
IO-145	61519de4	f6724182	140	140	7,365	1,242	2	61	1
IO-148	61519de4	f6724182	140	140	7,365	1,242	2	30	4
IO-153	61519de4	f6724182	140	140	7,365	1,242	6	56	1
IO-173	8de491fc	b1b9f1af	136	182	5,647	1,681	2	32	1
IO-275	8de491fc	b1b9f1af	136	182	5,647	1,681	2	1	1
IO-288	8de491fc	b1b9f1af	136	182	5,647	1,681	81	16	3
IO-290	8de491fc	b1b9f1af	136	182	5,647	1,681	2	5	1
IO-291	8de491fc	b1b9f1af	136	182	5,647	1,681	10	24	3
IO-297	8de491fc	b1b9f1af	136	182	5,647	1,681	9	13	1
IO-305	8de491fc	b1b9f1af	136	182	5,647	1,681	10	83	1

Index