Semantic Slicing of Software Version Histories

Yi Li, Chenguang Zhu, Julia Rubin, *Member, IEEE*, and Marsha Chechik, *Member, IEEE*

Abstract—Software developers often need to transfer functionality, e.g., a set of commits implementing a new feature or a bug fix, from one branch of a configuration management system to another. That can be a challenging task as the existing configuration management tools lack support for matching high-level, semantic functionality with low-level version histories. The developer thus has to either manually identify the exact set of semantically-related commits implementing the functionality of interest or sequentially port a segment of the change history, "inheriting" additional, unwanted functionality.

In this paper, we tackle this problem by providing automated support for identifying the set of semantically-related commits implementing a particular functionality, which is defined by a set of tests. We formally define the semantic slicing problem, provide an algorithm for identifying a set of commits that constitute a slice, and propose techniques to minimize the produced slice. We then instantiate the overall approach, CSLICER, in a specific implementation for Java projects managed in Git and evaluate its correctness and effectiveness on a set of open-source software repositories. We show that it allows to identify subsets of change histories that maintain the functionality of interest but are substantially smaller than the original ones.

Index Terms—Software changes, version control, dependency, program analysis.

1 Introduction

REAL software is seldom created "all at once", and changes are inevitable [2]. Software development is typically an incremental and iterative process where many program versions are created, each evolving and improving the previous ones. For example, new requirements, bugs and errors emerge during use and developers can take advantage of the knowledge and insights they gain to repair, enhance and optimize earlier versions of the system through incremental updates. This makes version history a crucial artifact in the software development process.

Software configuration management systems (SCM), such as Git [3], SVN [4] and Mercurial [5], are commonly used for hosting software development artifacts. They allow the developers to periodically submit their ongoing work, storing it as an increment over previous version. Such an increment is usually referred to as a *commit* (Git and SVN) or a *change set* (Mercurial), and we use these two terms interchangeably. Commits are stored sequentially and ordered by their time stamps, so that it is convenient to trace back to any version in the history. *Branching* is another construct provided by most modern SCM systems. Branches are used, for example, to store a still-in-development prototype version of a project or to store multiple project variants targeting different customers.

However, the sequential organization of changes is inflexible and lacks support for many tasks that require high-level, semantic understanding of program functionality [6], [7]. For example, developers often need to locate and transfer functionality, either for porting bug fixes between branches or for propagating features from development to release branches [8].

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

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 to the new branch as well, which is a tedious and error-prone manual task [9]. For example, consider the feature "make Groovy method blacklist truly append-only", introduced in version 1.3.8 of the Elasticsearch project [10] – a real-time distributed data search and analytics framework written in Java. This feature and its corresponding test case are implemented in a single commit (#`647327f4`). Yet, propagating this commit to a different branch will fail because one of the added statements makes use of a field whose declaration was introduced in an earlier commit (#`64d8e2ae`).

Including unwanted functionality and unnecessary commits in patches is often considered to be bad practice. For instance, most of the software projects implement strict guidelines of accepting only small and focused patches [11], [12], [13]. The main rationale behind these guidelines is to keep changes that have a different purpose separate, which leads to a speed up in the code review process, fewer merge conflicts, and easier future maintenance. For example, the Bitcoin Core [11] contributor guideline stresses the importance of simplicity of pull requests:

"Patchsets should always be focused. For example, a pull

- *Y. Li, C. Zhu, and M. Chechik are with the Department of Computer Science, University of Toronto, Toronto, ON, Canada, M5S3G4.*
  E-mail: {liyi, czhu, chechik}@cs.toronto.edu
- *J. Rubin is with the Department of Electrical and Computer Engineering, University of British Columbia, Vancouver, BC, V6T1Z4.*
  E-mail: mjulia@ece.ubc.ca

This article extends and improves the results presented in [1]. It contains novel techniques for history slice minimization, additional data in the experimental results and a more detailed review of the related work.

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."

In this paper, we look at the problem of identifying the exact minimal subset of history that implements a particular functionality of interest. Inspired by the concept of *program slicing* [14], we refer to this subset of semantically-related commits as a *semantics-preserving slice*. We assume that a functionality is defined by a set of tests exercising it. We propose a system CSLICER, which enhances the support provided by current SCM tools by mapping high-level functionalities to low-level commits.

CSLICER has two main phases: *semantic slicing* and *slice minimization*. The first phase consists of 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 relies on static and dynamic program 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 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, e.g., SVN or Git. This step can optionally be skipped when using language-aware merging tools [15] or in semantic-based SCM systems [16]. However, such systems are not dominant in practice yet.

The generic semantic slicing algorithm is conservative and can be imprecise. The second phase of CSLICER mitigates the imprecision and improves the quality of history slices through slice minimization. We investigate various sources of imprecision that commonly appear in practice and design several techniques to detect and remove the false positives. We first use a light-weight screening technique to filter out changes that are less likely to affect the target functionality according to heuristics. Then we enumerate all possible combinations of the remaining commits and find a minimal subset of the original history which preserves the functionality of interest. Empirical results show that our proposed slice minimization techniques can effectively improve the solution quality of CSLICER.

The use case of the CSLICER system is not limited to functionality porting; it can also be used for refactoring existing branches, e.g., by splitting them into functionality-related ones. We instantiate CSLICER for Java projects hosted in Git. To empirically evaluate the effectiveness and scalability of our approach, we experiment with a set of open source software projects. The results show that our approach can identify functionality-relevant subsets of original histories that (a) correctly capture the functionality of interest while being (b) minimal in many cases or (c) substantially smaller than the original ones.

Contributions. In our prior work [1], we presented an algorithm which computes an over-approximated semantic history slice and evaluated the prototype implementation on a few subjects subjects. In this paper, we propose novel slice minimization techniques on top of the original algorithm to improve quality of the computed semantic slices. We also extend and restructure the empirical studies to better evaluate the effectiveness of our approach. We summarize the contributions as follows.

- We formally define the *semantic slicing problem* for software version histories and propose a generic 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 propose a number of heuristic-based techniques which can effectively improve slice quality by reducing false positives and minimizing history slices.
- We instantiate the overall approach, CSLICER, by providing a fully automated 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. We compare our two-phase minimization technique with the state-of-art – delta debugging [17]. Our experiments show that CSLICER is able to correctly identify functionality-relevant minimal subsets of change histories more efficiently.

Organization. The rest of the paper is organized as follows. We start with a simple example in Section 2, illustrating CSLICER. It is followed by necessary background and definitions in Section 3. In Section 4, we formalize the semantic slicing algorithm and prove its correctness. In Section 5, we define the notion of minimal history slice and propose several techniques that can improve quality of semantic slices. In Section 6, we describe the implementation details and optimizations. In Section 7, we report on our case studies and empirical findings. Finally, in Section 8 and 9, we compare CSLICER with related work and conclude the paper, respectively.

2 CSLICER BY EXAMPLE

In this section, we illustrate CSLICER on a simple schematic example inspired by the feature migration case in the Elasticsearch project [10]. Figure 1 shows a fragment of the change history between versions v1.0 and v1.1 for the file `Foo.java`. 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 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 2 on the left.

Each dashed box in Figure 1 encloses a commit written in the *unified format* (the output of command `diff -u`). The

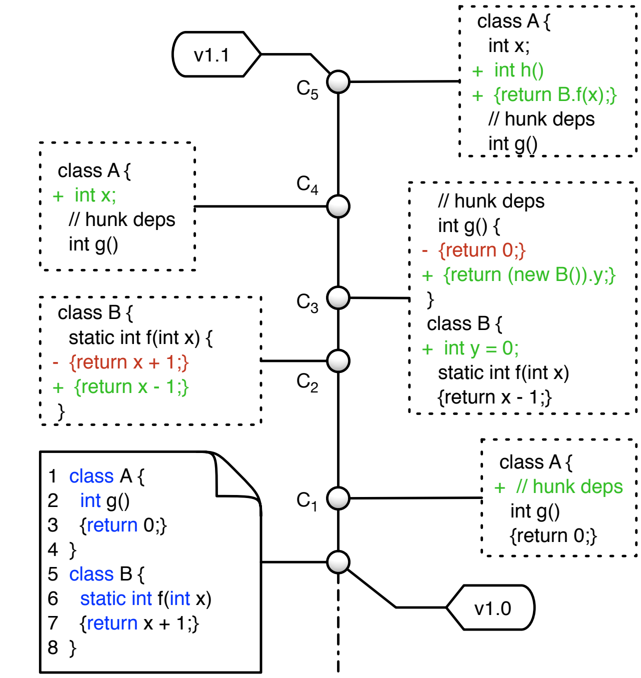


Fig. 1. Change history of `Foo.java`.

```
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;}
}
```

```
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;}
}
```

Fig. 2. `Foo.java` before and after semantic slicing.

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 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 the current example, the maximum length of the contexts is four lines: up to two lines before and after each change.

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 (1) the body of method `B.f` was changed in C_2 and the change is required to produce the correct result; (2) the declaration of field `A.x` was introduced in C_4 but was missing in v1.0, which would cause compilation errors; and (3) 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.

In fact, the change histories form a dependency hierarchy

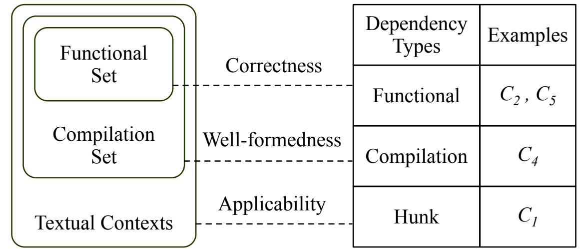


Fig. 3. Change dependency hierarchy.

with respect to the target functionality (see Figure 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`. 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. 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. 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.

Then in the slice minimization phase, CSLICER attempts to reduce from the identified commits and in this particular case fails since the solution is already optimal – there is no smaller set of commits that can preserve all of the desired properties. We investigate more complex cases in Section 5.

Applying the set of commits in sequence on top of v1.0 produces a new program shown in Figure 2 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.

3 Background

In this section, we provide the background needed in the rest of the paper.

3.1 Language Syntax

To keep the presentation of our algorithm 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* [18], denoting it by P. The syntax rules of the language P are given in Figure 4. 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

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

Fig. 4. Language syntax rules [18].

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

Fig. 5. Subtyping rules [18].

correctness are retained [19]. We discuss additional language features in Section 7.

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 *constructors* ($\overline{K}$). 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.

The subtyping rules of P, shown in Figure 5, 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 6) – field *overshadowing* and method *overloading* are not allowed while method *overriding* is allowed in Featherweight Java [18]. 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.

3.2 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 7 shows two ASTs for the program `Foo.java` before and after the change set C_3 is applied. In the left AST, the following facts are true about the node `f`,

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

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

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

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

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

Fig. 6. Fields and methods lookup [18].

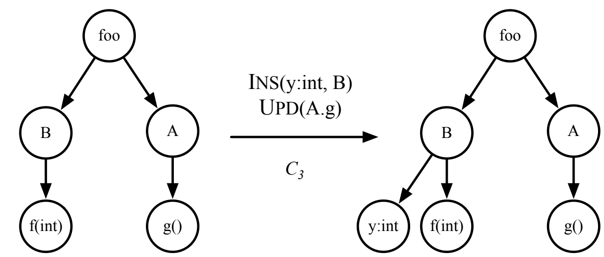


Fig. 7. Visualize C_3 as a sequence of atomic changes applied on ASTs.

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

3.3 Changes and Change Histories

Let Γ be the set of all ASTs. Now we define what change, change set and change history as AST transformation operations.

Definition 1. *(Atomic Change). An atomic change operation $\delta : \Gamma \to \Gamma$ is either an* insert, delete *or* update *(see Figure 8). It transforms $r \in \Gamma$ producing a new AST r' such that $r' = \delta(r)$.*

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.

Definition 2. *(Change Set). Let r and r' be two ASTs. A change set $\Delta : \Gamma \to \Gamma$ 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 [21]. For instance, in Figure 7, C_3 consists of an insertion of a new node `y` to `B` followed by an update of the node `g`.

Definition 3. *(Change History). A history of changes is a sequence of change sets, i.e., $H = \langle\Delta_1, \ldots, \Delta_k\rangle$.*

Definition 4. *(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.*

$$\frac{y \in V(r)}{V(r') \leftarrow V(r) \cup \{x\} \quad \textsc{Parent}(x) \leftarrow y} \textsc{Ins}((x, n, v), y)$$
$$id(x) \leftarrow n \quad \nu(x) \leftarrow v$$

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

Fig. 8. Types of atomic changes [20].

We write $H' \lhd H$ indicating H' is a sub-history of H and refer to $\langle \Delta_i, \ldots, \Delta_j \rangle$ as $H_{i..j}$. The applicability of a history is defined similar to that of change sets.

3.4 Test Cases

We assume that semantic functionalities can be captured by test cases and the execution trace of a test case is deterministic. For simplicity, a test case 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 5. *(Test Case). A test case t is a function $t : P \rightarrow \mathbb{B}$ such that for a given program $p \in P$, $t(p)$ is true if and only if the test succeeds, and false otherwise.*

A test suite is a collection of unit 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., $\forall t \in T \cdot t(p)$.

4 CSlicer Phase 1: Semantic Slicing

In this section, we define the semantic slicing problem and present our slicing algorithms in detail.

4.1 Overview of the Workflow

We start with the formal problem definition followed by a high level overview of our approach.

4.1.1 Problem Definition

Definition 6. *(Semantics-preserving Slice). 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., $p_k \models T$. 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) $H'(p_0) \models T$.

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

To address this problem, we devise an efficient algorithm which requires only a one-time effort for compilation and test execution, but may produce sub-optimal results. 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 [22].

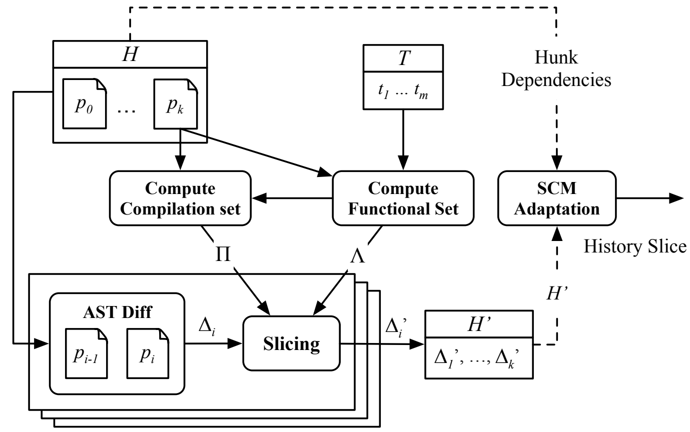


Fig. 9. High-level over view of the semantic slicing algorithms.

4.1.2 Workflow

Figure 9 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 $\langle \Delta_1, \ldots, \Delta_k \rangle$ 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 Adaption) 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 2.

Step 1: Computing Functional Set. CSlicer executes the test on the latest version of the program (left-hand side of Figure 2), 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 $\{$`A.h`, `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 later.

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 $\{$`A`, `A.x`, `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 1). Having computed the functional and compilation set (highlighted in Figure 2), 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.

4.2 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 9. The optional SCM adaptation phase will be discussed in Section 4.3.

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

Fig. 10. Algorithm 1: the semantic slicing algorithm.

4.2.1 Algorithm 1

The main SEMANTICSLICE procedure is shown in Figure 10. 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 explicitly initialized during declaration and all methods (and constructors) called during runtime.

COMPDEP(p_k, Λ)**.** The procedure COMPDEP analyzes *reference* relations in p_k and includes all referenced code entities of Λ into the compilation set Π. We borrow the set of rules for computing Π from Kästner and Apel [19], where the authors formally prove that their rules are complete and ensure that no reference without a target is ever present in a program. Applying these rules, which are given in Figure 11 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;

$$\frac{C <: D \quad C \in \Pi}{D \in \Pi}\,[\text{L.1}] \qquad \frac{f : C \in \Pi}{C \in \Pi}\,[\text{L.2}] \qquad \frac{C(\overline{D\ f})\{\texttt{super}(\overline{f});\overline{\texttt{this}.f = f};\} \in \Pi}{C \in \Pi \quad \overline{D} \in \Pi \quad \overline{f} \in \Pi}\,[\text{K1}]$$

$$\frac{C\ \ m(\overline{D\ x})\{\texttt{return}\ e;\} \in \Pi}{C \in \Pi \quad \overline{D} \in \Pi}\,[\text{M1}] \qquad \frac{\ldots\{\texttt{return}\ e.f;\} \in \Pi}{f \in \Pi}\,[\text{E1}] \qquad \frac{\ldots\{\texttt{return}\ e.m(\overline{e});\} \in \Pi}{m \in \Pi}\,[\text{E2}]$$

$$\frac{\ldots\{\texttt{return new}\ C(\overline{e});\} \in \Pi}{C \in \Pi}\,[\text{E3}] \qquad \frac{\ldots\{\texttt{return}\ (C)e;\} \in \Pi}{C \in \Pi}\,[\text{E4}] \qquad \frac{x \in \Pi}{\textsc{Parent}(x) \in \Pi}\,[\text{P1}] \qquad \frac{x \in \Lambda}{x \in \Pi}\,[\text{T1}]$$

Fig. 11. CompDep reference relation rules.

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 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,

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

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

4.2.2 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 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 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 $id(\delta)$ itself or its parent should present (Figure 8). Because of the 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 $\delta \in H'$ stays 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 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 6, 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 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 [15]. As an optional step, SCM adaptation integrates the generic Algorithm 1 with text-based SCM systems such as Git.

4.3.1 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.

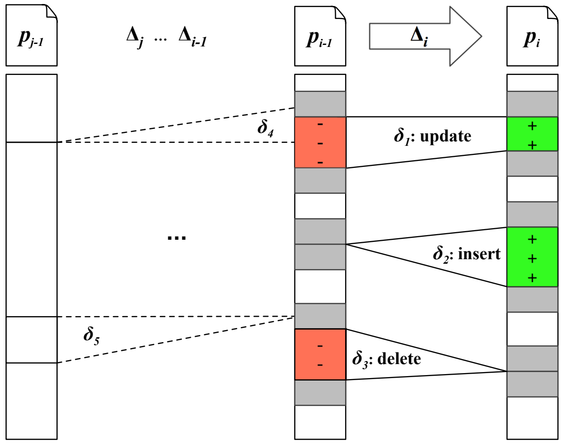


Fig. 12. Illustration of direct Hunk dependencies.

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** δ is DEL $\wedge\ right(\delta) \in range(B_i)$ **then**
6: $D \leftarrow D \cup \Delta_{i-1}$
7: **else if** δ is INS $\wedge\ right(\delta) \cap B_i \neq \emptyset$ **then**
8: $D \leftarrow D \cup \Delta_{i-1}$
9: $B_{i-1} \leftarrow B_{i-1}/right(\delta)$
10: **end for**
11: $D \leftarrow D \cup$ DIRECTHUNK($B_{i-1}, H_{1..(i-1)}$)
12: **return** D
13: **end procedure**

Fig. 13. Algorithm 2: the DIRECTHUNK procedure.

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.

4.3.2 Calculating Hunk Dependencies

Algorithm 1 treats changes between versions as tree edit operations. Another view of changes used by text-based SCM tools is called *hunk*. 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 12 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 change 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 change set Δ_i *directly hunk-depends* on another change set 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* [23], i.e., reordering them in history does not cause conflict. The procedure HUNKDEP(H') returns the transitive hunk dependencies for all change set in H', i.e.,

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

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

Given a change set 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 12). Figure 13 describes the algorithm for computing the set of direct hunk dependencies ($\rightsquigarrow$) by tracing back in history and locating the latest change sets that contribute to each line of the basis. Starting from Δ_{i-1}, we iterate backwards through all preceding change sets. If a change set 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 12, $\Delta_i \rightsquigarrow \Delta_j$ 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 $O(|H'| \times |H| \times \max_{\Delta \in H}(|\Delta|))$.

5 CSLICER PHASE 2: SLICE MINIMIZATION

The problem of finding optimal semantics-preserving sub-history is intractable in general, as we showed in Section 4.1.1. However, there are many cases where a minimal sub-history is preferred or even required. For example, when submitting pull requests for review, contributors should refrain from including unrelated changes as suggested by many project contribution guidelines [11], [12], [13]. Also, developers commonly suggest to split a mixed patch into multiple ones, e.g., as indicated by these code review comments:

> *"Okay, nevertheless we need to split this up because it is unrelated to the issue we're talking about."*[1]

Therefore, we would like to produce logically clean and easy-to-merge history slices by reducing all irrelevant changes. Despite the complexity of finding shortest slices, we have identified a number of heuristic-based techniques that could help shorten history slices and possibly derive

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

```
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`

Fig. 14. Example illustrating different types of false positives.

minimal ones. In this section, we define *minimal semantic slice* and discuss a few such techniques for minimizing history slices obtained from Phase 1.

5.1 Minimal Semantic Slice

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 6).

Definition 7. *(Minimal Semantic Slice). Given a semantics-preserving slice H^* such that $H^* \lhd_T H$. H^* is a minimal semantic slice of H if, $\forall H_{sub} \lhd H^* \cdot (|H_{sub}| < |H^*|) \implies \neg(H_{sub} \lhd_T H)$.*

For our running example in Section 2, the solution produced by CSLICER is not only minimal but also optimal as there does not exist any other sub-history which is semantics-preserving. However, a minimal semantic slice does not always correspond to the global optimal slice. In other words, there might exist a shorter semantic slice H_{opt} which is not a sub-history of H^*. Empirical evidences show that minimal slice of a semantics-preserving slice (such as H' returned by Algorithm 1) is a good approximation to H_{opt} (see Section 7.2.2).

5.2 Sources of Imprecision

The CSLICER algorithm (Algorithm 1) presented in Section 4 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 (details in Section 7) which can be divided into two groups, namely (1) semantics-preserving changes, and (2) oracle unobservable changes.

Figure 14 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.

Semantics-Preserving Changes. An example of semantics-preserving changes is code refactoring [24]. Refactoring changes 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. [25], 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 [25] which gives developers the flexibility to replay the refactoring changes after merging is done. Therefore, the slice minimization phase aims to produce logically clean and easy-to-merge history slices by isolating all semantics-preserving changes.

Renaming of code entities is one commonly seen refactoring change. In fact, existing code refactoring detection techniques [26], [27], [28] 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 only consider self-contained local and low-level refactorings [29] 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 [30] change history, we observe the adoption of new Java 7 features `try-with-resources` statement [31] and the diamond operator [32] 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 [33] test case fails either due to assertion failures (explicit checks defined by developers) or runtime errors (implicit checks performed by runtime system) [34]. Some changes, even if may alter the program behaviors, are non-observable to the test oracle. 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 behaviors.

5.3 Techniques for Slice Minimization

As discussed in Section 4.1.1, enumerating all sub-histories of a history H is highly unrealistic when the length of H is

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(Δ) **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:
11: **procedure** ENUMERATE(p_0, H_{S2}, T)
12: $L \leftarrow Sort$(PICKABLE(HUNKGRAPH(H_{S2})))
13: **for all** $H_{sub} \in L$ **do**
14: **if** $H_{sub}(p_0) \models T$ **then return** H_{sub}
15: **return** H_{S2} ▷ no shorter history found
16: **end procedure**

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

Fig. 15. Algorithm 3: finding minimal semantic slice of a given history H.

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 Figure 15. 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 returns a valid slice with best-effort.

5.3.1 Static Pattern Matching (S1)

The AST differencing algorithm used in Algorithm 1 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* [20]. Similar to the definition in Fluri and Gall [20], 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 with predefined patterns.

Some examples of the patterns include local refactoring/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.

5.3.2 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 [14] is able to identify more precisely the set of program statements (a program slice) which have the potential to affect the test oracle. However, a program slice does not always subsume an oracle observable set since a change outside of the program slice can still be executed and affect the test results.

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 are 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 7.2.2). The procedure PICKABLE in Figure 15 uses an approach similar to the Kahn's topological sorting algorithm [35] to generate all cherry-pickable proper sub-histories for a given history slice.

The idea behind the procedure PICKABLE (Lines 17-27) 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 20). 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 $\bigcup_{r \in \text{ROOTNODES}(V_{k+1}, E_{k+1})}\{\overline{V_{k+1}/r}\} \cup$ PICKABLE$(V_{k+1}/r, E_{k+1}/Out(r))$. □

Theorem 2. *(Correctness of Algorithm 3).* 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[2]. The theorem trivially holds given Lemma 3 and the fact that the sub-histories are traversed in increasing order of their lengths. □

Algorithm 3 always terminates since there are only finitely many sub-histories. However, in practice, enumerating all combinations can still be time consuming even with the pre-processing step. We report our empirical findings in Section 7.2.2.

6 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.

6.1 Implementation

Figure 16 shows the high-level architecture of our CSLICER implementation. We implemented CSLICER in Java, using the JaCoCo Java Code Coverage Library [36] for byte code instrumentation and collecting execution data during runtime. We modified the Java source code change extraction tool ChangeDistiller [37] for AST differencing and change classification. We also used the Apache Byte Code Engineering Library (BCEL) [38] for entity reference relation analysis. The hunk dependency detection component

2. This assumption can be relaxed in semantic-based SCM systems.

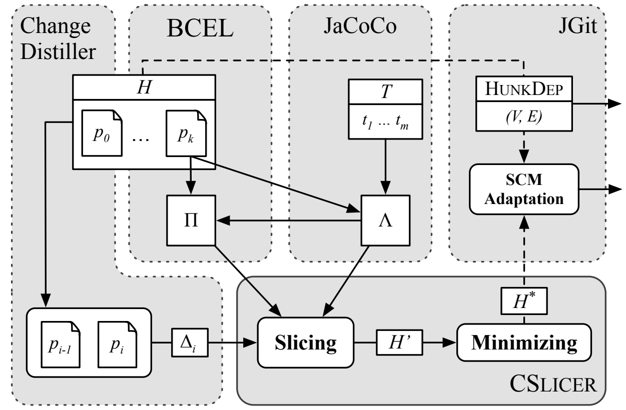


Fig. 16. Architecture of the CSLICER implementation.

HUNKDEP was developed based on the Java-based Git implementation, JGit [39]. 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 [30]. 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.

6.2 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.

6.2.1 *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 [40]. 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.

6.2.2 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.

6.2.3 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.

7 Evaluation

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

1) Qualitative assessment of CSlicer in practical settings (Section 7.1). 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 7.2). 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.
3) Comparison with the state-of-the-art change minimization technique – delta debugging [17] (Section 7.3). We tested CSlicer and delta debugging end-to-end and compared the quality of the produced slices as well as performance of the two systems.

7.1 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 [41], Elasticsearch [10], Apache Maven [30], Apache Commons Collection [42], Apache Commons Math [43], and Apache Commons IO [44].

TABLE 1
Statistics of case study subjects. Each row lists the number of Java files (#Files), lines of code (LOC) of the studied projects, the length of the chosen history fragment ($|H|$), the number of changed files (f), lines added (+), and lines deleted (-) for the chosen range, and the number of test cases ($|T|$) in the target test suites.

Case	Project	#Files	LOC	$\|H\|$	Changed f	Changed +	Changed -	$\|T\|$
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

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.

Statistics of the subjects are given in Table 1 grouped by their case numbers (Column "Case"). 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.

7.1.1 Case 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. 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 17). This was all done automatically as a preprocessing step. The expanded feature branch has 267 commits in total.

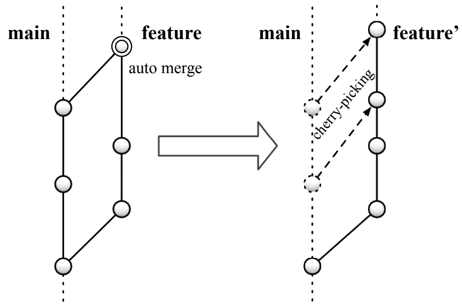


Fig. 17. Illustration of expanding auto merges through cherry-picking corresponding commits from the main branch to the feature branch.

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.

7.1.2 Case 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, 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 without using minimization 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 the effectiveness of slice minimization techniques on reducing such false alarms in Section 7.2.

7.1.3 Case 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 (see the quote of code review comments in Section 5). 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 2
Pull requests recreating results. The Columns "ID" and "Status" show the ID and status of the corresponding pull requests. The Column "$|H^*|$" shows the length of the pull requests created and submitted by the developers. The Columns "$|H'|$", "$P(\%)$", and "$R(\%)$" list the length, precision, and recall of the pull requests created by CSLICER, respectively.

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[3] from four different software projects (see Table 2). 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 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 exact same results as the ones submitted by the developers. For Commons IO, CSLICER included one extra commit (#bccf2af4) 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 an empty merge commit (#4cc49d78) in pull request #7 which was included by the developer.

7.2 Quantitative Evaluation of CSLICER

To have more insights of the internals of our algorithm, we also empirically evaluated the efficiency and applicability

3. 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.

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:

RQ1: How effective is the CSLICER *slicing algorithm*, in terms of the number of irrelevant commits identified?
RQ2: How efficient is CSLICER when applied to *histories of various lengths*?
RQ3: How effective are the slice *minimization techniques* used in CSLICER?

7.2.1 Experiment 1: History Slicing

The first experiment aims to address both **RQ1** and **RQ2** 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 [45]. 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 five; 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. For example, the Maven development guidelines [46] state that:

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

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 3.[4] 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 [8]. 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.

4. A more detailed version of the subject descriptions is available at: http://www.cs.toronto.edu/~polaris/tse/data.html.

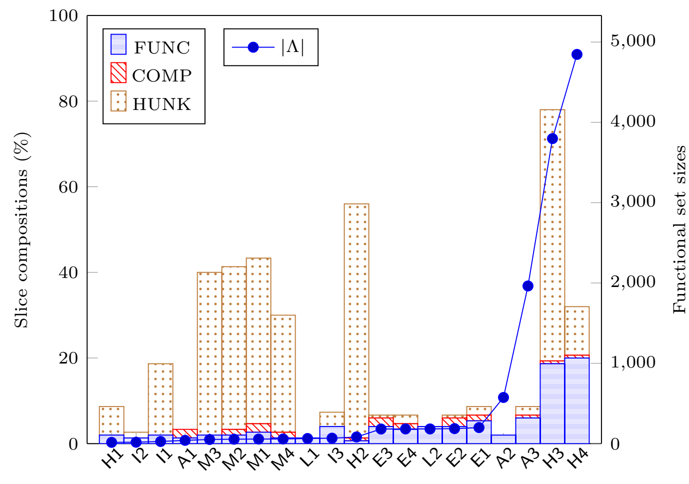


Fig. 18. Compositions of sliced histories for all benchmarks shown in the order of increasing functional set sizes. Left y-axis represents percentage of functional (FUNC), compilation (COMP) and hunk (HUNK) dependencies over $|H|$ (%). Right y-axis represents sizes of the functional set ($|\Lambda|$).

Results. The slice compositions for medium-length histories are reported in Figure 18. The history slices returned by CSLICER consist of functional, compilation and hunk dependencies. Each stacked bar in Figure 18 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.

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 RQ1. CSLICER effectively reduces irrelevant commits given a target test suite.

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 19. 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.

TABLE 3
Target functionalities used in the experiments. 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 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.

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 Toolchains-BuildingException	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

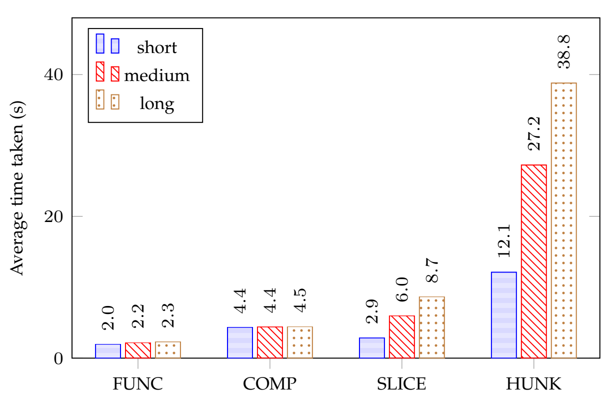


Fig. 19. Average time taken by CSLICER when running on short, medium and long histories. Each vertical bar plots the average time taken by the functional set computation (FUNC), compilation set computation (COMP), core slicing component (SLICE) and the hunk dependency computation (HUNK) when running on histories with different lengths.

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 RQ2. CSLICER scales well on real-world software projects and histories of moderate lengths.

7.2.2 Experiment 2: Minimization

The second experiment is designed to answer **RQ3** by running CSLICER 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 1, 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

TABLE 4
Results for Experiment 2. Columns "$|H'|$" and "$|H^*|$" show the lengths of slices produced in Experiment 1 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.

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

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 lengths of slices before and after minimization are given in Table 4.

For each of the subjects, we first applied static pattern matching (S1) to identify and eliminate change sets with low significance. This includes 2 local code refactorings, 7 white list statement updates, and 4 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 the 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 3 for details).

Results. The detailed experimental results are reported in Table 4. 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 at minimal lengths by enumerating all their candidate sub-histories with 3, 6 and 3 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, 6 commits were filtered out during the S1 step for E2. In addition, applying S1 significantly reduced the number of sub-histories needed 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.

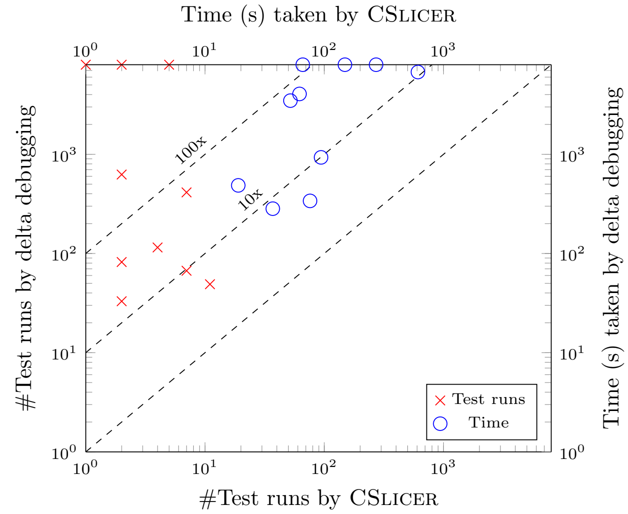


Fig. 20. End-to-end comparison results of delta debugging and CSlicer. The left and right y-axes represent the number of tests and the total running time required by delta debugging to find a minimal solution, respectively. The bottom and top x-axes represent the number of tests and the total running time required by CSlicer to finish, respectively. All axes use the log scale.

Answer to RQ3. 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.

7.3 Comparison with Delta Debugging

Delta debugging [17] is an automatic debugging technique developed by Andreas Zeller in 1999. Its goal is to simplify and isolate a minimal cause of a given test failure. The high-level idea is to repeatedly partition the input and opportunistically reduce the search space when the target test fails on one of the partitions until a minimal partition is reached. This problem can be considered to be 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 CSlicer on 10 benchmarks for which we have established the ground truth (see Table 4). We looked at both the total running time and the number of tests required to reach the minimal solution.

The detailed results are shown in Figure 20. CSlicer performs better than delta debugging for points above the 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, CSLICER 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 CSLICER's win is that repeated test executions are rather expensive and Phase 1 of CSLICER can effectively reduce the search space. Moreover, Phase 1 is relatively inexpensive, taking only 37.4% of the total running time. Overall, CSLICER performs constantly better than delta debugging, with the majority of the cases seeing a 10X to 100X speedup.

7.4 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 clean pull requests. The comparison with manually identified optimal sub-histories indicates that the precision of Algorithm 1 is limited by how accurately we can decide whether a change affects the test results. Further evaluation showed that our proposed slice minimization techniques are effective for slice quality improvement and thus are good complements to Algorithm 1. 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. Finally, from the comparison with the delta-debugging-style minimization approach, we highlight the benefits of the two-phase design of CSLICER. By combining the inexpensive but over-approximating Phase 1 with the heavyweight but precise Phase 2, CSLICER achieves both precision and efficiency.

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.

Experiment 2 was limited to subjects whose initial slice is relatively short so that we could exhaustively enumerate all sub-histories and find a minimal slice. The false positive rates of CSLICER may not be generalizable when applied on other experimental subjects.

8 Related Work

To the best of our knowledge, the software history semantic slicing problem was first defined in our prior work [1] and it has not been studied in the literature prior to that or since then. However, our work does intersect with different areas spanning history understanding and manipulation, code change classification, change impact analysis and software product line variants generation. We compare CSLICER with the related work below.

8.1 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 [29], [47], localize bugs [17], [48], and support predictions [49], [50]. Delta debugging [17] uses divide-and-conquer-style iterative test executions to narrowing down the causes of software failures. It has been applied to minimize the set of changes which cause regression test failures. However, in contrast to CSLICER which extracts semantic information from successful test runs, there is no easy way to exploit failed executions. Regarding slice quality, Zeller and Hildebrandt [48] consider an approximated version of minimality, i.e., *1-minimal*, which only 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. In contrast, Algorithm 3 relies on semantic information and much cheaper heuristic-based change filtering techniques to first shorten the input before exhaustive enumeration so that true minimality becomes tractable. Algorithm 3 guarantees the quality of the minimized slice – removing any sub-history from the slice breaks the target properties.

Another interesting take on history analysis is *history transformation* [7], [51]. Muşlu et. al [7] 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 [7] is a much more meaningful representation for history understanding and analysis.

8.2 Change Classification

The CSLICER algorithm relies on sophisticated structural differencing [21], [52], [53] and code change classification [20], [37], [54] 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 the atomic changes according to their change types.

The most established AST differencing algorithm is ChangeDistiller [37]. 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.

8.3 Change Impact Analysis

Change Impact Analysis [40], [55], [56], [57], [58] solves the problem of determining the effects of source code modifications. It usually means selecting a subset of tests from a regression test suite that might be affected by the given change, or, given a test failure, deciding which changes might be causing it.

Research on impact analysis can be roughly divided into three categories: the *static* [55], [59], *dynamic* [56] and *combined* [40], [58], [60] approaches. The work most related is on the combined approaches to change impact analysis. Ren et. al [40] introduced a tool, Chianti, for change impact analysis of Java programs. Chianti takes two versions of a Java program and a set of tests as the input. First, it builds dynamic call graphs for both versions before and after the changes through test execution. Then it compares the classified changes with the old call graph to predict the affected tests; and it uses the new call graph to select the affecting changes that might cause the test failures. FaultTracer [58] improved Chianti by extending the standard dynamic call graph with field access information.

CSLICER uses similar techniques to identify affecting changes. However, the real challenge in our problem is to process and analyze the identified affecting changes and ensure that all related dependencies are included as well. Moreover, we consider a sequence of program versions rather than only two versions, and our algorithm can operate on both the atomic change level and the text-based commit level.

8.4 Product Line Variant Generation

The software product line (SPL) [61], [62] community faces similar challenges as we do. An SPL is an efficient means for generating a family of program variants from a common code base [19], [63]. Code fragments can be disabled or enabled based on user requirements, e.g., using "`#ifdef`" statements in C, often resulting in ill-formed programs. Therefore, variant generation algorithms need to check the implementation of SPL to ensure that the generated variants are well-formed.

Kästner et. al [63] 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 [19] proposed an extended calculus for reasoning about the type-soundness of product line variant programs written in Featherweight Java. 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 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.

9 Conclusion and Future Work

In this paper, 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 and addressed them using a combination of static matching and dynamic enumeration techniques. 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 scale in practical applications.

We see many avenues for future work. First, it is useful to extend CSLICER in order to handle distributed change histories where a special treatment for branching and merging operations is needed. Second, semantic slicing is a part of our larger goal to enable history-aware feature "copy and paste" – transferring functional units to arbitrary branches within the same repository. Semantic slicing is essential for finding and extracting existing software functionalities from change histories. A natural next step is to investigate the possibility of merging the extracted history slices with other code bases. The biggest challenge here is to precisely detect both syntactic and semantic conflicts that might occur during slice integration and search for correct fixes automatically or interactively. Finally, another interesting direction is to integrate the CSLICER algorithm with language-aware merge tools and investigate possible trade-offs.

References

[1] Y. Li, J. Rubin, and M. Chechik, "Semantic Slicing of Software Version Histories," in *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, Lincoln, NE, USA, November 2015, pp. 686–696.

[2] I. Sommerville, *Software Engineering (7th Edition)*. Pearson Addison Wesley, 2004.

[3] Git Version Control System. [Online]. Available: https://git-scm.com
[4] Apache Subversion (SVN) version control system. [Online]. Available: http://subversion.apache.org
[5] Mercurial Source Control Management System. [Online]. Available: http://mercurial.selenic.com
[6] J. Rubin, A. Kirshin, G. Botterweck, and M. Chechik, "Managing Forked Product Variants," in *Proc. of SPLC'12*, 2012, pp. 156–160.
[7] K. Muşlu, L. Swart, Y. Brun, and M. D. Ernst, "Development History Granularity Transformations," in *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, Lincoln, NE, USA, November 2015, pp. 697–702.
[8] Y. Li, C. Zhu, J. Rubin, and M. Chechik, "Precise Semantic History Slicing through Dynamic Delta Refinement," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, Singapore, Singapore, September 2016, pp. 495–506.
[9] J. Rubin, A. Kirshin, G. Botterweck, and M. Chechik, "Managing Forked Product Variants," in *Proceedings of the 16th International Software Product Line Conference*, vol. 1. New York, NY, USA: ACM, 2012, pp. 156–160.
[10] Elasticsearch: Distributed, Open Source Search and Analytics Engine. [Online]. Available: https://www.elastic.co/products/elasticsearch
[11] How to Contribute Code to Bitcoin Core. [Online]. Available: https://bitcoincore.org/en/faq/contributing-code
[12] How to Get Your Change Into the Linux Kernel. [Online]. Available: https://www.kernel.org/doc/Documentation/SubmittingPatches
[13] Microsoft Azure: Ways to Contribute. [Online]. Available: https://azure.github.io/guidelines
[14] F. Tip, "A Survey of Program Slicing Techniques," *Journal of Programming Languages*, vol. 3, pp. 121–189, 1995.
[15] The Diff and Merge Tool that Understands Your Code – SemanticMerge. [Online]. Available: https://www.semanticmerge.com
[16] Cow: Semantic Version Control. [Online]. Available: http://jelv.is/cow
[17] A. 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*. London, UK, UK: Springer-Verlag, 1999, pp. 253–267.
[18] A. Igarashi, B. C. Pierce, and P. Wadler, "Featherweight Java: A Minimal Core Calculus for Java and GJ," *ACM Transactions on Programming Languages and Systems*, vol. 23, no. 3, pp. 396–450, May 2001.
[19] C. Kästner and S. Apel, "Type-Checking Software Product Lines - A Formal Approach," in *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 258–267.
[20] B. Fluri and H. C. Gall, "Classifying Change Types for Qualifying Change Couplings," in *Proceedings of the 14th IEEE International Conference on Program Comprehension*. IEEE, 2006, pp. 35–45.
[21] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom, "Change Detection in Hierarchically Structured Information," in *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, 1996, pp. 493–504.
[22] G. Rothermel and M. J. Harrold, "A Framework for Evaluating Regression Test Selection Techniques," in *Proceedings of the 16th International Conference on Software Engineering*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1994, pp. 201–210.
[23] (2016, March) Understanding Darcs/Patch Theory. [Online]. Available: http://en.wikibooks.org/wiki/Understanding_Darcs/Patch_theory
[24] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley, 1999.
[25] D. Dig, K. Manzoor, R. E. Johnson, and T. N. Nguyen, "Effective Software Merging in the Presence of Object-Oriented Refactorings," *IEEE Transactions on Software Engineering*, vol. 34, no. 3, pp. 321–335, May 2008.
[26] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson, "Automated Detection of Refactorings in Evolving Components," in *Proceedings of the 20th European Conference on Object-Oriented Programming*. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 404–428.
[27] C. Gorg and P. Weissgerber, "Detecting and visualizing refactorings from software archives," in *Proceedings of the 13th International Workshop on Program Comprehension*, 2005.
[28] D. Kawrykow and M. P. Robillard, "Non-essential Changes in Version Histories," in *Proceedings of the 33rd International Conference on Software Engineering*. New York, NY, USA: ACM, 2011, pp. 351–360.
[29] E. Murphy-Hill, C. Parnin, and A. P. Black, "How We Refactor, and How We Know It," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 5–18, Jan 2012.
[30] Apache Maven Project. [Online]. Available: https://maven.apache.org
[31] (2016, March) The try-with-resources Statement. [Online]. Available: https://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html
[32] Type Inference for Generic Instance Creation. [Online]. Available: https://docs.oracle.com/javase/8/docs/technotes/guides/language/type-inference-generic-instance-creation.html
[33] A Unit Testing Framework for the Java Programming Language. [Online]. Available: http://junit.org
[34] D. Schuler and A. Zeller, "Assessing Oracle Quality with Checked Coverage," in *Proceedings of the 4th IEEE International Conference on Software Testing, Verification and Validation*. Washington, DC, USA: IEEE Computer Society, 2011, pp. 90–99.
[35] A. B. Kahn, "Topological Sorting of Large Networks," *Communications of the ACM*, vol. 5, no. 11, pp. 558–562, Nov. 1962.
[36] JaCoCo Java Code Coverage Library. [Online]. Available: http://www.eclemma.org/jacoco
[37] B. Fluri, M. Wuersch, M. PInzger, and H. Gall, "Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction," *IEEE Transactions on Software Engineering*, vol. 33, no. 11, pp. 725–743, Nov. 2007.
[38] Apache Byte Code Engineering Library. [Online]. Available: https://commons.apache.org/proper/commons-bcel
[39] JGit: A Lightweight, Pure Java Library Implementing the Git Version Control System. [Online]. Available: https://eclipse.org/jgit
[40] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. 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*. New York, NY, USA: ACM, 2004, pp. 432–448.
[41] Apache Hadoop Project. [Online]. Available: https://hadoop.apache.org
[42] "Commons Collections: The Apache Commons Collections Library," https://commons.apache.org/proper/commons-collections.
[43] "Commons Math: The Apache Commons Mathematics Library," https://commons.apache.org/proper/commons-math.
[44] "Commons IO: The Apache Commons IO Library," https://commons.apache.org/proper/commons-io.
[45] Apache ActiveMQ. [Online]. Available: http://activemq.apache.org
[46] Developing Maven. [Online]. Available: http://maven.apache.org/guides/development/guide-maven-development.html
[47] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin, "Early Detection of Collaboration Conflicts and Risks," *IEEE Transactions on Software Engineering*, vol. 39, no. 10, pp. 1358–1375, Oct. 2013.
[48] A. Zeller and R. Hildebrandt, "Simplifying and Isolating Failure-inducing Input," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 183–200, 2002.
[49] T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller, "Mining Version Histories to Guide Software Changes," in *Proceedings of the 26th International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 563–572.
[50] K. Herzig and A. Zeller, "The Impact of Tangled Code Changes," in *Proceedings of the 10th Working Conference on Mining Software Repositories*. Piscataway, NJ, USA: IEEE Press, 2013, pp. 121–130.
[51] S. Hayashi, T. Omori, T. Zenmyo, K. Maruyama, and M. Saeki, "Refactoring Edit History of Source Code," in *Proceedings of the 28th IEEE International Conference on Software Maintenance*. IEEE, September 2012, pp. 617–620.
[52] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Montperrus, "Fine-grained and Accurate Source Code Differencing," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, Sep. 2014, pp. 313–324.
[53] P. Bille, "A Survey on Tree Edit Distance and Related Problems," *Theoretical Computer Science*, vol. 337, no. 1-3, pp. 217–239, Jun. 2005.
[54] M. Hashimoto and A. Mori, "Diff/TS: A Tool for Fine-Grained Structural Change Analysis," in *Proceedings of the 15th Working*

Conference on Reverse Engineering. Antwerp: IEEE, October 2008, pp. 279–288.

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

[56] J. Law and G. Rothermel, “Whole Program Path-Based Dynamic Impact Analysis,” in *Proceedings of the 25th International Conference on Software Engineering*. IEEE, May 2003, pp. 308–318.

[57] S. Zhang, Z. Gu, Y. Lin, and J. Zhao, “Change impact analysis for AspectJ programs,” in *Proceedings of the 24th IEEE International Conference on Software Maintenance*, September 2008, pp. 87–96.

[58] L. Zhang, M. Kim, and S. Khurshid, “Localizing Failure-inducing Program Edits Based on Spectrum Information,” in *Proceedings of the 27th International Conference on Software Maintenance*. IEEE, 2011, pp. 23–32.

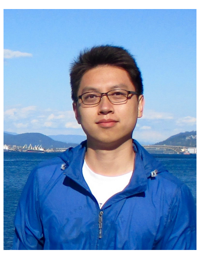
[59] D. C. Kung, J. Gao, P. Hsia, F. Wen, Y. Toyoshima, and C. Chen, “Change Impact Identification in Object Oriented Software Maintenance,” in *Proceedings of the International Conference on Software Maintenance*. Washington, DC, USA: IEEE Computer Society, 1994, pp. 202–211.

[60] A. Orso, T. Apiwattanapong, and M. J. 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*. New York, NY, USA: ACM, 2003, pp. 128–137.

[61] P. C. Clements and L. Northrop, *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001.

[62] K. Pohl, G. Boeckle, and F. van der Linden, *Software Product Line Engineering : Foundations, Principles, and Techniques*. Springer, 2005.

[63] C. Kästner, S. Apel, S. Trujillo, M. Kuhlemann, and D. 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)*, ser. LNBI, vol. 33. Zurich, Switzerland: Springer Berlin Heidelberg, June 2009, pp. 175–194.



Yi Li is a PhD candidate in the Department of Computer Science at the University of Toronto. Yi received his B.Comp. degree with First Class Honors from the National University of Singapore in 2011 and the M.Sc. degree in Computer Science from the University of Toronto in 2013. His research interests include program analysis, software verification, software requirements and history analysis. His research also addressed important problems in SMT solving techniques and artificial intelligence. His recent work on software history analysis won an ACM Distinguished Paper Award at the 30th International Conference on Automated Software Engineering (ASE’15). Yi Li also served as a member of the Artifact Evaluation Committee of the 28th International Conference on Computer Aided Verification (CAV’16).

Chenguang Zhu received the B.E. degree in software engineering from the Harbin Institute of Technology in 2015. He is working toward the M.Sc. degree in Computer Science and is a research assistant working in the Software Engineering group in the Department of Computer Science at the University of Toronto. His current research interests include software version history analysis, verification, and testing.

Julia Rubin is an Assistant Professor at department of Electrical and Computer Engineering at the University of British Columbia. She received her PhD in Computer Science from the University of Toronto and both M.Sc. and B.Sc. degrees in Computer Science from the Technion. During the 2014-2016 academic years, Julia was a postdoctoral researcher in the department of Electrical Engineering and Computer Science at MIT. Earlier, she spent more than 10 years in industry, working for a startup company and then for the IBM Research Lab in Israel, where she was a research staff member and, part of the time, a research group manager. Julia’s research interests are in software engineering, program analysis, software security, and software sustainability, focusing on topics related to ¡¡¡¡¡¡¡ paper.tex the construction of reliable software in an efficient manner. ======= the construction of reliable software in an efficient manner. ¿¿¿¿¿¿¿ 1.19 Julia serves as a member of the program committee for several major conferences ¡¡¡¡¡¡¡ paper.tex in Software Engineering. She was co-chair of the program committees of SPLC 2014, ECMFA in 2014, will serve as a co-chair of FASE 2017, MODELS Doctoral Symposium in 2017, and ICSE Doctoral ======= in Software Engineering. She was co-chair of the program committees of SPLC’14, ECMFA’14, will serve as a co-chair of FASE’17, MODELS Doctoral Symposium in 2017, and ICSE Doctoral ¿¿¿¿¿¿¿ 1.19 Symposium in 2018. Julia also served as a member-at-large of the TCSE Executive Committee from 2013 to 2016, and chaired the TCSE Distinguished Women in Science and Engineering Leadership Award Committee.

Marsha Chechik is Professor and Bell University Labs Chair in Software Engineering in the Department of Computer Science at the University of Toronto. Prof. Chechik’s research interests are in modeling and reasoning about software. She has authored over 100 papers in formal methods, software specification and verification, computer security and requirements engineering. Marsha Chechik has been Program Committee Co-Chair of a number of conferences in verification (TACAS’16, VSTTE’16, CONCUR’08) and software engineering (ASE’14, FASE’09, CASCON’08), and is gearing up to co-chair the technical program committee of the 2018 International Conference on Software Engineering (ICSE’18). She has been fortunate to work with many extremely talented graduate students and postdocs, some of whom are now conducting research in top universities in Canada, the US, Chile, Luxembourg, and China.