Semantic Slicing of Software Version Histories

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

Julia Rubin
Massachusetts Institute of Technology
Cambridge, MA, USA
mjulia@csail.mit.edu

Marsha Chechik
University of Toronto
Toronto, ON, Canada
chechik@cs.toronto.edu

***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 specific subset 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 refer to our approach, CSLICER, as semantic slicing of version histories. We formally define the semantic slicing problem, provide an algorithm for identifying a set of commits that constitute a slice, and instantiate it in a specific implementation for Java projects managed in Git. We evaluate the correctness and effectiveness of our approach 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.

***Keywords*—software changes; version history; dependency.**

I. INTRODUCTION

Software configuration management systems (SCM), such as Git [1], SVN [2] and Mercurial [3], 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. *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.

Occasionally, developers need to migrate a specific functionality – e.g., a feature or a bug fix – from one branch to another. *Back-porting* is one example of such a migration, when changes made in a newer version of software are ported to an earlier one in order to provide the updated functionality to all users. Several SCM systems provide mechanism of "replaying" commits on a different branch, e.g., the `cherry-pick` command in Git. Yet, little support is provided for matching high-level functionality with commits that implement it: SCM systems only keep track of temporal and text-level dependencies between the managed commits. The job of identifying the exact set of commits implementing the functionality of interest is left to the developers.

Even in very disciplined projects, when such commits can be identified by browsing their associated log messages, the functionality of interest might depend on another functionality implemented in the same branch. To ensure correct execution, that other functionality has to be identified and migrated to the new branch as well, which is a tedious and error-prone manual task [4]. For example, consider the feature "make Groovy method blacklist truly append-only", introduced in version 1.3.8 of the Elasticsearch project [5] – 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`).

In this paper, we look at this problem in detail. Borrowing the *program slicing* [6] concept, we define a functionality-specific *semantic history slice* as a subset of semantically related commits from the original history that preserve the functionality of interest. 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 systems by mapping high-level functionalities to low-level commits.

CSLICER has two main phases. The first phase is generic and independent of any specific SCM system in use. It 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 second phase adapts the output produced in the first one to specifics of SCM systems. More precisely, it maps the collected set of atomic changes back to the commits in the original change history. It also takes care of merge conflicts that can occur when cherry-picking commits in text-based SCM systems, e.g., SVN or Git. This phase can optionally be skipped when using language-aware merging tools [7] or in semantic-based SCM systems [8]. However, such systems are not dominant in practice yet.

The use case of the CSLICER system is not limited to functionality porting only; 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 allows to identify functionality-relevant subsets of original histories that (a) correctly capture the functionality of interest while being (b) substantially smaller than the original ones.

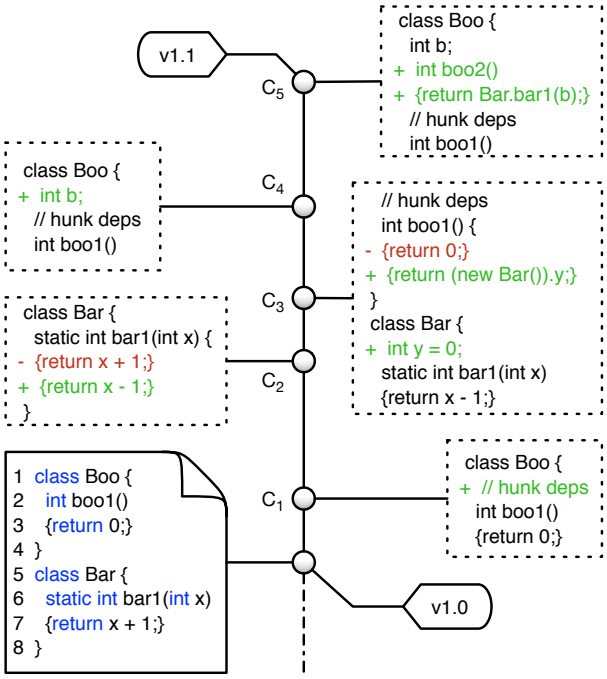

Fig. 1. Change history of `Foo.java`.

Contributions. We summarize our 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 instantiate the overall approach, CSLICER, by providing a fully-automatic semantic slicing tool applicable for Git projects implemented in Java. The source code of the tool, as well as binaries and examples used in this paper, are available at https://bitbucket.org/liyistc/gitslice.
- We evaluate the tool on a number of real-world medium- to large-scale software projects. Our experiments show that CSLICER is able to correctly identify functionality-relevant subsets of change histories that are substantially smaller than the original ones.

Organization. We start with a simple example in Sec. II, illustrating CSLICER. In Sec. III, we provide the necessary background and definitions. In Sec. IV, we formalize the semantic slicing approach and prove its correctness. In Sec. V, we describe implementation details and report on our experimental findings. Finally, in Sec. VI and VII, we compare CSLICER with related work and conclude the paper, respectively.

II. CSLICER BY EXAMPLE

In this section, we illustrate CSLICER on a simple schematic example inspired by the feature migration case in the Elasticsearch project [5]. Fig. 1 shows a fragment of the change history between versions v1.0 and v1.1 for the file `Foo.java`.

```
1  class Boo {
2    int b;
3    int boo2()
4    {return Bar.bar1(b);}
5    // hunk deps
6    int boo1()
7    {return (new Bar()).y;}
8  }
9  class Bar {
10   int y = 0;
11   static int bar1(int x)
12   {return x - 1;}
13 }
```

```
class Boo {
  int b;
  int boo2()
  {return Bar.bar1(b);}
  // hunk deps
  int boo1()
  {return 0;}
}
class Bar {
  static int bar1(int x)
  {return x - 1;}
}
```

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

Initially, as shown in version v1.0, the file contains two classes, `Boo` and `Bar`, each having a member method `boo1` and `bar1`, respectively.

Later, in change set C_1, a line with a textual comment was inserted right before the declaration of `Boo.boo1`. Then, in change set C_2, the body of `Bar.bar1` was modified from `{return x+1;}` to `{return x-1;}`. In change set C_3, the body of `Boo.boo1` was updated to return the value of a newly added field `y` in class `Bar`. In change set C_4, a field declaration was inserted in class `Boo` and, finally, in change set C_5, a new method `boo2` was added in class `Boo`. The resulting program in v1.1 is shown in Fig. 2 on the left.

Each dashed box in Fig. 1 encloses a commit written in the *unified format* (the output of command `diff -u`). The lines starting with "+" are inserted while those starting with "−" are deleted. Each bundle of changed lines is called a *hunk* and comes with a *context* – a certain number of lines of surrounding text that stay unchanged. In Fig. 1, these are the lines which do not start with "+" or "−". The context that comes with a hunk is useful for ensuring that it 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 `Boo.boo2` 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 `Bar.bar1` was changed in C_2 and the change is required to produce the correct result; (2) the declaration of field `Boo.b` 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.

To produce a syntactically and semantically correct result, CSLICER examines the program at version v1.1 by running the test that triggers `Boo.boo2` and verifies that it returns -1. CSLICER identifies the set of elements that directly participate in the test execution, i.e., methods `Boo.boo2` and `Bar.bar1` We call this set of elements a *functional set*. CSLICER then collects elements referenced by the functional set, i.e., classes `Boo`, `Bar` and the field declaration `int b` in `Boo`. We call this set a *compilation set*. Next, the algorithm searches for origins of the functional and compilation set elements in the history, identifying commits $\{C_2, C_4, C_5\}$. Finally, to calculate

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

Fig. 3. Language syntax rules [9].

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

Fig. 4. Subtyping rules [9].

the *hunk set*, for each of the identified commits, CSLICER recursively searches for text-level dependencies. That process identifies commit C_1 since C_5 depends on it. The algorithm outputs the set of all identified commits that are required for porting the functionality of interest to v1.0 successfully: $\{C_1, C_2, C_4, C_5\}$. More precise details about this process are given in Sec. IV.

Applying the identified set of commits in a sequence on top of v1.0 produces a new program shown in Fig. 2 on the right. It is easy to verify that the call to `Boo.boo2` in both programs returns the same value. Changes introduced in commit C_3 – an addition of the field `Bar.y` and a modification of the method `Boo.boo1` – do not affect the test results and are not part of any other commit context. Thus, this commit can be omitted.

III. BACKGROUND

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

Language Syntax. To keep the presentation of our algorithm concise, we step back from the complexities of the full Java language and concentrate on the core object-oriented features. We adopt a simple functional subset of Java from *Featherweight Java* [9], denoting it by P. The syntax rules of the language P are given in Fig. 3. Many advanced Java features, e.g., interfaces, abstract classes and reflection are stripped from P, while the typing rules which are crucial for the compilation correctness are retained [10]. We discuss additional language features in Sec. V.

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. 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 Fig. 4, 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 Fig. 5) – field *overshadowing* and method *overloading* are not allowed while method *overriding* is allowed in Featherweight Java [9].

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

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

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

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

Fig. 5. Fields and methods lookup [9].

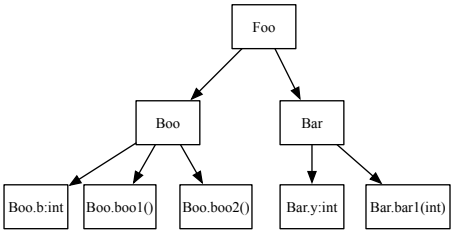

Fig. 6. AST of `Foo.java` at v1.1.

Abstract Syntax Tree. 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 $\textsc{Parent}(x)$. Fig. 6 shows an AST for the program `Foo.java` at version v1.1.

Change, Change Set and History. Let Γ be the set of all ASTs. An *atomic change* operation $\delta : \Gamma \to \Gamma$ is either an *insert*, *delete* or *update* (see Fig. 7). 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 node x with node 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.

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 [12].

A *history* of changes is a sequence of change sets $H = \langle\Delta_1, \ldots, \Delta_k\rangle$. A *sub-history* is a sub-sequence of a history, i.e. a sequence derived by removing change sets from H without altering the ordering. We write $H' \lhd H$ indicating H' is a sub-history of H and refer to $\langle\Delta_i, \ldots, \Delta_j\rangle$ as $H_{i..j}$. The applicability of a history is defined similar to that of change sets.

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

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

Fig. 7. Types of atomic changes [11].

Test Cases. We assume that semantic functionality can be captured by test cases and the execution trace of a test case is deterministic. Let T be a set of test cases $\{t_i\}$. We write $p \models T$ if program p passes all tests in T, i.e., $p \models t$ for all $t \in T$.

IV. The CSlicer System

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

A. Overview of CSlicer Workflow

Problem Definition. 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., $p_k \models T$. Our goal is to (conservatively) identify a sub-history $H' \lhd H$ such that the following properties hold:

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

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.

As such, 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 has been shown to be undecidable [13].

There are two main phases in our CSlicer approach: semantic slicing and SCM adaptation. Fig. 8 illustrates the high-level workflow of the semantic slicing phase. 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 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. 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'. Below we describe each step in turn, illustrating them through the example in Sec. II.

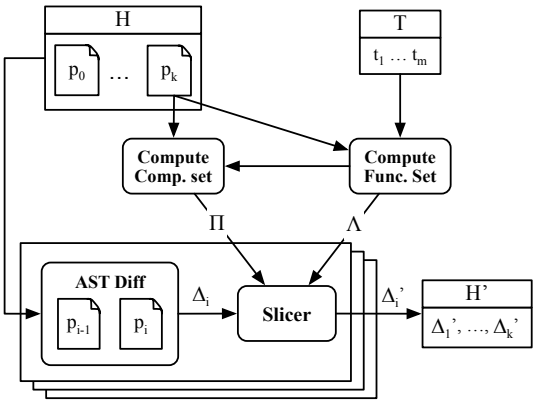

Fig. 8. The semantic slicing phase workflow.

Compute Functional Set. First, CSlicer executes the test on the latest version of the program (left-hand side of Fig. 2), which triggers method `Boo.boo2`. It dynamically collects the program statements traversed by this execution. These include the method bodies of `Boo.boo2` and `Bar.bar1`. 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 $\{$`Boo.boo2`, `Bar.bar1`$\}$. 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.

Compute 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 $\{$`Boo`, `Boo.b`, `Bar`$\}$. Notice that the classes `Boo` and `Bar` are included as well since the fields and methods require their enclosing classes to be present.

Core Slicer Component. In the core 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* [14]. Hav-

ing computed the functional and compilation set (highlighted in Fig. 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 – `Bar.bar1` and `Boo.boo2` – are in the functional set. C_4 contains an insertion of `Boo.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.

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 Sec. IV-C.

B. Semantic Slicing Algorithm

The main SEMANTICSLICE procedure is shown in Fig. 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 [10], where the authors formally prove that their set of rules is complete and ensures that no reference without a target is ever present in a program. Applying these rules, which are given in Fig. 9 and described below, allows us to guarantee type safety of the sliced program.

- L1 a class can only extends a class that is present;
- L2 a field can only have type of a class that is present;
- K1 a constructor can only have parameter types of classes that are present and access to fields that are present;
- M1 a method declaration can only have return type and parameter types of classes that are present;
- E1 a field access can only access fields that are present;
- E2 a method invocation can only invoke methods that are present;
- E3 an instance creation can only create objects from classes that are present;
- E4 a cast operation can only cast an expression to a class that is present;
- P1 an entity is only present when the enclosing entities are present;
- T1 an entity is in the compilation set if it is in the functional set.

We iterate backwards through all the change sets in the history (Lines 6-20) 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,

$$\text{LOOKUP}(\delta, p) \triangleq \exists m, C \cdot \text{METHODS}(m, C) \neq \text{METHODS}'(m, C),$$

where METHODS and METHODS$'$ are the method lookup function for p and $\delta(p)$, respectively. Finally, the sliced history H' is returned at Line 21.

Correctness. 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 Line 10 and 13, 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 (Fig. 7). Because of the COMPDEP rule (P1), i.e., $x \in \Pi \Rightarrow \text{PARENT}(x) \in \Pi$, 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

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

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

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

Fig. 9. CompDep reference relation rules.

```
1:  procedure SemanticSlice(p_0, H, T)
2:      H', k ← ⟨⟩, |H|                          ▷ initialization
3:      p_k ← H(p_0)                             ▷ p_k is the latest version
4:      Λ ← FuncDep(p_k, T)                      ▷ functional set
5:      Π ← CompDep(p_k, Λ)                      ▷ compilation set
6:      for i ∈ [k, 1] do                        ▷ iterate backwards
7:          Δ'_i ← ⟨⟩                            ▷ initialize sliced change set
8:          for δ ∈ Δ_i do
9:              if ¬Lookup(δ, H_{1..i}(p_0)) then    ▷ keep lookup
10:                 if δ is Del ∨ id(δ) ∉ Π then
11:                     continue                 ▷ skip non-comp and deletes
12:                 end if
13:                 if δ is Upd ∧ id(δ) ∉ Λ then
14:                     continue                 ▷ skip non-test updates
15:                 end if
16:             end if
17:             Δ'_i ← Δ'_i, δ                   ▷ concatenate the rest
18:         end for
19:         H' ← H', Δ'_i                        ▷ grow H'
20:     end for
21:     return H'
22: end procedure
```

Fig. 10. The semantic slicing algorithm.

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. *Soundness. 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 set of test cases such that $p_k \models T$. Then the following properties hold for the sliced history $H' =$ SemanticSlice(p_0, H, T):*

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

Proof: 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 6-20) of the algorithm. 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 the functional set, (3) also holds. Thus, $H'(p_0)$ satisfies (1), (2) and (3). ■

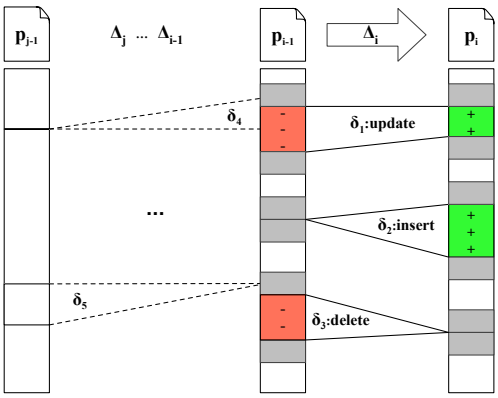

Fig. 11. Hunk dependency detection.

C. SCM Adaptation

Eliminating Side-Effects. The proposed algorithm operates on the atomic change level and can directly be used with semantic-based tools, such as SemanticMerge. Yet, 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 $H' = \langle \Delta'_1, \ldots, \Delta'_k \rangle$ (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 Δ'_i can thus be mapped to its counterpart in the original history, i.e., Δ_i.

However, original commits may contain changes that are sliced away by the code slicing algorithm. 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 the corresponding code entities back to their original state.

Calculating Hunk Dependencies, HunkDep(H'). The algorithm so far 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, Fig. 11 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,

```
1: procedure DIRECTHUNK(B_i, H_{1..i})
2:     D ← ∅
3:     B_{i-1} ← L_i
4:     for δ ∈ Δ_{i-1} do
5:         if δ is DEL ∧ right(δ) ∈ range(B_i) then
6:             D ← D ∪ Δ_{i-1}
7:         else if δ is INS ∧ right(δ) ∩ B_i ≠ ∅ then
8:             D ← D ∪ Δ_{i-1}
9:             B_{i-1} ← B_{i-1}/right(δ)
10:        end if
11:    end for
12:    D ← D ∪ DIRECTHUNK(B_{i-1}, H_{1..(i-1)})
13:    return D
14: end procedure
```

Fig. 12. The DIRECTHUNK procedure.

i.e., δ_1, δ_2 and δ_3. We define two auxiliary functions, $left(\delta)$ and $right(\delta)$, which return the lines 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* [15], 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.,

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

Once a sub-history H' is computed based on the functional and the compilation set, 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 Fig. 11). Fig. 12 describes the algorithm for computing the set of direct hunk dependencies ($\rightsquigarrow$) by tracing back 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 Fig. 11, $\Delta_i \rightsquigarrow \Delta_j$ 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 $O(|H'| \times |H| \times \max_{\Delta \in H}(|\Delta|))$.

V. IMPLEMENTATION AND EVALUATION

In this section, we describe the implementation of our tool, CSLICER, and report on the experiments evaluating its effectiveness and scalability.

A. Implementation

We have implemented CSLICER in Java, using the JaCoCo Java Code Coverage Library [16] for byte code instrumentation and execution data collection, and a modified version of the ChangeDistiller [17] for AST differencing as well as change type classification. We also use the Apache Byte Code Engineering Library (BCEL) [18] for entity reference relation analysis. The hunk dependency detection component HUNKDEP can also be used as a stand-alone tool for any text-based SCM system, e.g., Git. For a given set of commits, HUNKDEP generates a hunk dependency graph which can be used to reorganize commit history without causing conflicts.

Our CSLICER implementation works with Java projects hosted in Git repositories. The test-slice-verify process is fully-automatic for projects built with Maven [19]. For other build environments, user is required to manually build and collect test execution data through Jacoco plugins. When the analysis is finished, CSLICER automatically cherry-picks the identified commits and verifies the test results. The source code of CSLICER is made available online at https://bitbucket.org/liyistc/gitslice.

Handling Advanced Java features. We presented our algorithm 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 [14]. To address this, we treat class hierarchy changes as an update of 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.

Handling Non-Java changes. Real software version histories often contain changes to non-Java files, e.g., build scripts, configuration files and binaries libraries. Sometimes changes to non-Java files are mixed with Java 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 test behaviors.

Optimizations. To make the technique more configurable, we allow users to specify packages, files, classes and methods to include or exclude during both the analysis and cherry-picking processes. For example, all changes on test files (which are not part of the target system) are ignored by default. Similarly, changes on internal debugging code can also be discarded

TABLE I. CASE STUDY SUBJECTS.

Project	#Files(Java)	LOC	#C	Changed			#T
				f	+	-	
Hadoop	6608(5861)	1291K	267	1197	111119	14064	58
Elasticsearch	4412(3865)	616K	51	75	1755	304	2
CSLICER	119(112)	16K	94	132	12640	2383	1

without affecting the observable system bahavior. We noticed in the experiments that user domain knowledge about the projects can enhance the precision of the computed results. Another easy optimization is when a commit and its revert are detected in the history, we can safely ignore the pair without affecting the correctness of the approach.

B. Evaluation: Case Studies

We have conducted case studies and thoroughly investigated the results produced by CSLICER, to better understand its applicability, effectiveness, and limitations. In particular, we looked at three software projects of varying sizes, different version control workflows, and disparate committing styles – the Apache Hadoop project [20], the Elasticsearch project and our own tool, CSLICER. 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 about the studied subjects,

1) a new feature "write to single replica in memory" (ticket number HDFS-6581) added to Apache Hadoop;
2) a Groovy interface enhancement in Elasticsearch; and
3) a new feature that converts bytecode descriptors to fully qualified class identifier in CSLICER

are given in Table I. The "#C" column lists the number of commits within the chosen fragment of history. The "Changed" column lists the number of files changed (f), lines added (+) and lines deleted (-) for the chosen range. All of the following studies were conducted on a computer running Linux with an Intel i5 3.1GHz processor and 4GB of RAM. Now we describe each case study in detail.

Hadoop. 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, sometimes 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 functionality 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. 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 a equivalent expanded linear history (see Fig. 13). The expanded feature branch has 267 commits in total.

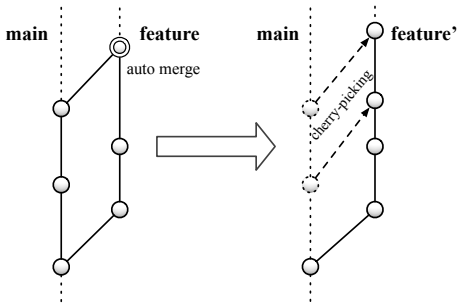

Fig. 13. Illustration of expanding auto merges.

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 compilation dependency as well as preserving the test behaviors, and additional 26 commits for hunk dependency. Note that some commits from the main branch are actually required by the target feature. The refactored feature branch contains 91 commits in total, which achieves ∼66% reduction.

Elasticsearch. As discussed in Sec. IV, there is no efficient algorithm that returns optimal solution in general. From our experience, finding the shortest functionality 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 and the next subject.

We took a fragment of history between v1.3.6 to v1.3.8 of Elasticsearch, which includes the total of 51 commits. There are 2 unit tests clearly marked by the developers intending to test the target functionality. CSLICER identified 17 commits achieving a 67% reduction of the unrelated commits. However, compared with the optimal solution which requires 4 commits, CSLICER reports 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 are intended to be verified. For instance, the test case "`testDynamicBlacklist`" invokes not only the components implementing the "dynamic black list" but also those that implement the logging functions for debug purposes. Obviously, changes on the logging functions do not affect the test results. But without prior knowledge, CSLICER would conservatively classify them as possibly affecting changes. This is one of the limitations of our current technique.

However, as mentioned earlier, it is not always possible to decide whether an arbitrary change affects the test results in a given program. Two possible options can help alleviate this problem. The first one is allowing users with domain knowledge to provide insights on which components (including

classes, methods, fields and statements) do not affect the test results. Given accurate information, we can prune the functional set and reduce false positives. The other option is to analytically answer the question for some specific type of tests through more sophisticated program analysis techniques, e.g., data-flow analysis [21], differential symbolic execution [22] or differential static analysis [23]. We leave it for future work.

CSLICER. We repeated the similar study on the CSLICER repository. With expertise on our own project, we were able to exclude console output related methods from the functional set. This single exclusion helped reduce the number of test- and compilation-related commits from 5 down to 3. An interesting observation is that many more commits are required for hunk dependency in our repository (35/94 compared with 26/267 in Hadoop and 0/51 in Elasticsearch), which makes the reduction about 60%.

The two open source projects have much larger code bases and are relatively better managed than the CSLICER repository. In both Hadoop and Elasticsearch repositories, each commit is required, by an external contribution guideline [24], to be a stand-alone logical unit. For example, changes to different software components are usually separated into multiple commits; formatting, refactoring and documentation changes are separated from functional ones. Pairs of commits are more likely to depend on each other if the changes are less disciplined. Thus, we conjecture that the reduction rate in the CSLICER repository can be increased by the improved committing style.

C. Evaluation: Applicability

We also empirically evaluated the applicability of CSLICER by measuring its performance and the history reduction rate achieved when applied on real-world software projects. Specifically, we aimed to answer the following research questions:

RQ1: How *efficient* is CSLICER when applied on histories of various scales?

RQ2: What is the *history reduction rate*, in terms of the ratio of irrelevant commits, achieved by CSLICER?

Subjects and Methodology. We selected four open source software projects written in Java: three projects described above and, in addition, the Maven project. From each project, we randomly chose three functionalities (e.g., a feature, an enhancement and a bug fix) that are accompanied by good documentations and unit tests. It is often possible to link specific commits with on-line issue tracking documentations via ticket numbers embedded in the commit messages. For each functionality, we referred to the log messages and ticket numbers to locate the target commits where the functionality was introduced. The set of tests are either explicitly mentioned in the accompanied test plan or implicitly enclosed within the same commit as the functionality itself.

Then we took three sets of histories of length 50 (short), 100 (medium) and 150 (long) respectively, 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. We then verified that the sliced sources compile successfully and pass the same tests.

TABLE II. EXPERIMENTAL RESULTS.

F	Reduction(%)			Time(s) for Long	
	Short(H)	Medium(H)	Long(H)	Slice	Hunk
C1	94(62)	89(34)	91(28)	<1	1.1
C2	36(14)	39(10)	42(12)	<1	34.8
C3	82(20)	72(13)	71(14)	<1	129.7
E1	72(72)	79(78)	81(79)	2677.5	11.6
E2	94(90)	96(92)	95(91)	2086.4	12.8
E3	94(94)	95(94)	96(94)	2041.0	12.4
H1	52(24)	61(25)	53(21)	852.0	60.5
H2	50(44)	56(50)	67(59)	766.1	53.2
H3	88(84)	87(75)	90(71)	734.4	23.3
M1	94(90)	97(64)	97(59)	11.1	38.3
M2	96(96)	97(80)	98(79)	8.4	11.5
M3	94(94)	93(89)	95(92)	7.1	1.0
Avg.	79(65)	80(59)	81(58)	765.6	32.5

Results. The experimental results are reported in Table II. The column "F" lists the subject functionality where "C" stands for CSLICER, "E" stands for Elasticsearch, "H" stands for Hadoop HDFS (a sub-project of Hadoop) and "M" stands for Maven. The history reduction rate is shown separately for each set of histories. The numbers in brackets are the reduction with hunk dependencies counted. The time taken (for the longest history) by the main algorithm and the hunk dependency algorithm is shown in the "Slice" and "Hunk" columns, respectively.

The table shows that CSLICER achieves good reduction rate in most of the cases, especially for Elasticsearch. Interestingly, Elasticsearch also has few hunk dependencies. Majority of the analysis time was taken in the COMPDEP procedure for detecting reference relations from the bytecode. There are $\sim$6500 and $\sim$4200 classes for Elasticsearch and Hadoop HDFS respectively, which took long for processing. The computed reference graphs can be cached for reuse. The time taken by the HUNKDEP procedure is approximately linear to the number of commits remained after slicing.

Threats to Validity. The reduction rate 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.

Summary. To summarize, we evaluated CSLICER through both case studies and experiments. We demonstrated that apart from assisting developers in porting functionality, CSLICER can also be applied for branch refactoring. The comparison with manually identified optimal sub-histories indicates that the precision of our results is limited by how accurately we can decide whether a change affects the test results. We proposed some options for improving that. Yet, the results of quantitative studies suggest that CSLICER is able to achieve good reduction of commits in real world software repositories, which justifies its value in practice.

VI. RELATED WORK

To the best of our knowledge, the software history semantic slicing problem we defined in this paper is not previously studied in the literature. However, our work does intersect with different areas spanning code change classification, change

impact analysis, and software product line variants generation. We compare CSLICER with these related work below.

Change Classification. The CSLICER algorithm relies on sophisticated structural differencing [12], [25], [26] and code change classification [11], [17], [27] 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 [17]. It uses statement as the smallest AST node 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, move operation is no longer needed and never reported in CSLICER. We also label each AST node using a unique identifier to represent the fully qualified name of a 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.

Change Impact Analysis. Change Impact Analysis [14], [28]–[31] 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 the changes that might be causing it. The research on impact analysis can be roughly divided into three categories: the *static* [28], [32], *dynamic* [29] and *combined* [14], [31], [33] approaches.

The most related work in change impact analysis are the combined approaches. Ren et. al [14] 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 [31] 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.

The problem of finding a minimal subset of the history that contributes to a test failure also appeared in *delta debugging* [34]. The high-level idea is similar to what is implemented in the `git-bisect` [35] command, where a divide-and-conquer approach is used to locate a problematic commit through repeated test runs. Since the test result may not be a monotonic property with respect to the length of the history, such techniques cannot guarantee to find optimal solution within $O(\log k)$ test runs. Nevertheless, they can still be applied in conjunction with CSLICER to further reduce the solution size.

Product Line Variant Generation. The software product line (SPL) [36], [37] 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 [10], [38]. 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 [38] 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 [10] 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 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.

VII. CONCLUSION

We proposed CSLICER, an efficient semantic slicing algorithm which lives 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 it results in a syntactically correct and well-typed program. The computed semantic slice also captures the interested functional behaviors 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. Our prototype tool demonstrates its efficiency and achieves significant reduction when applied to large scale software projects.

We see many avenues for future work. First, a natural next step is to raise the precision of the functional set computation using more sophisticated static analysis techniques. We believe the reduction rate can be further improved with more accurate test affecting analysis. Another interesting direction is to integrate CSLICER algorithm with language-aware merge tools and investigate possible tradeoffs.

REFERENCES

[1] Git version control system. [Online]. Available: https://git-scm.com/

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

[3] Mercurial source control management system. [Online]. Available: http://mercurial.selenic.com/

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

[5] Elasticsearch: distributed, open source search and analytics engine. [Online]. Available: https://github.com/elastic/elasticsearch

[6] F. Tip, "A Survey of Program Slicing Techniques," *J. of Programming Languages*, vol. 3, pp. 121–189, 1995.

[7] The diff and merge tool that understands your code – SemanticMerge. [Online]. Available: https://www.semanticmerge.com

[8] Cow: Semantic Version Control. [Online]. Available: http://jelv.is/cow

[9] A. Igarashi, B. C. Pierce, and P. Wadler, "Featherweight Java: A Minimal Core Calculus for Java and GJ," *ACM Trans. Program. Lang. Syst.*, vol. 23, no. 3, pp. 396–450, May 2001.

[10] C. Kästner and S. Apel, "Type-Checking Software Product Lines - A Formal Approach," in *Proc. of ASE'08*, 2008, pp. 258–267.

[11] B. Fluri and H. C. Gall, "Classifying Change Types for Qualifying Change Couplings," in *Proc. of ICPC'06*, 2006, pp. 35–45.

[12] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom, "Change Detection in Hierarchically Structured Information," in *Proc. of SIGMOD'96*, 1996, pp. 493–504.

[13] G. Rothermel and M. J. Harrold, "A Framework for Evaluating Regression Test Selection Techniques," in *Proc. of ICSE'94*, 1994, pp. 201–210.

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

[15] Understanding Darcs/Patch Theory. [Online]. Available: http://en.wikibooks.org/wiki/Understanding_Darcs/Patch_theory

[16] JaCoCo Java Code Coverage Library. [Online]. Available: http://www.eclemma.org/jacoco

[17] B. Fluri, M. Wuersch, M. PInzger, and H. Gall, "Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction," *IEEE Trans. Softw. Eng.*, vol. 33, no. 11, pp. 725–743, Nov. 2007.

[18] Apache Byte Code Engineering Library. [Online]. Available: https://commons.apache.org/proper/commons-bcel

[19] Apache Maven Project. [Online]. Available: https://maven.apache.org

[20] Apache Hadoop Project. [Online]. Available: https://hadoop.apache.org

[21] M. Sagiv, T. Reps, and S. Horwitz, "Precise Interprocedural Dataflow Analysis with Applications to Constant Propagation," in *Proc. of TAPSOFT'95*, 1996, pp. 131–170.

[22] S. Person, M. B. Dwyer, S. Elbaum, and C. S. Păsăreanu, "Differential Symbolic Execution," in *Proc. of SIGSOFT FSE'08*, 2008, pp. 226–237.

[23] S. K. Lahiri, K. Vaswani, and C. A. Hoare, "Differential Static Analysis: Opportunities, Applications, and Challenges," in *Proc. of FOSER'00*, 2010, pp. 201–204.

[24] How to Contribute to Hadoop Common. [Online]. Available: https://wiki.apache.org/hadoop/HowToContribute

[25] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Montperrus, "Fine-grained and Accurate Source Code Differencing," in *Proc. of ASE'14*, ser. ASE'14, 2014, pp. 313–324.

[26] P. Bille, "A Survey on Tree Edit Distance and Related Problems," *Theor. Comput. Sci.*, vol. 337, no. 1-3, pp. 217–239, Jun. 2005.

[27] M. Hashimoto and A. Mori, "Diff/TS: A Tool for Fine-Grained Structural Change Analysis," in *Proc. of WCRE'08*, 2008, pp. 279–288.

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

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

[30] S. Zhang, Z. Gu, Y. Lin, and J. Zhao, "Change impact analysis for AspectJ programs," in *Proc. of ICSM'08*, Sept 2008, pp. 87–96.

[31] L. Zhang, M. Kim, and S. Khurshid, "Localizing Failure-inducing Program Edits Based on Spectrum Information," in *Proc. of ICSM'01*, 2011, pp. 23–32.

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

[33] A. Orso, T. Apiwattanapong, and M. J. Harrold, "Leveraging Field Data for Impact Analysis and Regression Testing," in *Proc. of ESEC/FSE'11*, 2003, pp. 128–137.

[34] A. Zeller and R. Hildebrandt, "Simplifying and Isolating Failure-inducing Input," *IEEE Trans. Softw. Eng.*, vol. 28, no. 2, pp. 183–200, 2002.

[35] git-bisect: Find by binary search the change that introduced a bug. [Online]. Available: http://git-scm.com/docs/git-bisect

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

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

[38] C. Kästner, S. Apel, S. Trujillo, M. Kuhlemann, and D. Batory, "Guaranteeing Syntactic Correctness for all Product Line Variants: A Language-Independent Approach," in *Proc. of TOOLS EUROPE'09*, ser. LNBI, vol. 33, 2009, pp. 174–194.