Noname manuscript No.
(will be inserted by the editor)

Precise Semantic History Slicing Through Dynamic Delta Refinement

Yi Li · Chenguang Zhu · Milos Gligoric · Julia Rubin · Marsha Chechik

Received: date / Accepted: date

Abstract Semantic history slicing solves the problem of extracting changes related to a particular high-level functionality from software version histories. State-of-the-art techniques combine static program analysis and dynamic execution tracing to infer an over-approximated set of changes that can preserve the functional behaviors captured by a test suite. However, due to the conservative nature of such techniques, the sliced histories may contain irrelevant changes. In this paper, we propose a divide-and-conquer-style partitioning approach enhanced by dynamic delta refinement to produce much smaller semantic history slices. We utilize deltas in dynamic invariants generated from successive test executions to learn significance of changes with respect to the target functionality. Additionally, we introduce a file-level commit splitting technique for untangling unrelated changes introduced in a single commit. Empirical results indicate that these measurements accurately rank changes according to their relevance to the desired test behaviors and thus partition history slices in an efficient and effective manner.

Y. Li
School of Computer Science and Engineering
Nanyang Technological University, Singapore, 639798
E-mail: yi_li@ntu.edu.sg

C. Zhu · M. Gligoric
Department of Electrical and Computer Engineering
University of Texas at Austin, TX, USA, 78712
E-mail: {cgzhu, gligoric}@utexas.edu

J. Rubin
Department of Electrical and Computer Engineering
University of British Columbia, Vancouver, BC, V6T1Z4
E-mail: mjulia@ece.ubc.ca

M. Chechik
Department of Computer Science
University of Toronto, ON, Canada, M5S3G4
E-mail: chechik@cs.toronto.edu

Keywords Semantic history slicing · Program analysis · Software configuration management

1 Introduction

Software Configuration Management systems (SCMs) are widely used in software development practices. These systems, e.g., Git [15], SVN [48], and Mercurial [35], are useful for capturing incremental changes made by developers, examining or reverting changes, identifying developers responsible for a specific change, creating development streams, and more. Incremental changes are manually grouped by developers to form *commits* (a.k.a. *change sets*). Commits are stored sequentially and ordered by their time stamps, so that it is convenient to trace back to any version in the software history.

Yet, the sequential organization of changes is inflexible and lacks support for many tasks that require high-level, semantic understanding of program functionality [36, 45]. For example, developers often need to locate and transfer functionality from one branch to another: either for porting bug fixes or for splitting large chunk commits into multiple functionally-independent pull requests. Developers also face the challenge of identifying failure-inducing changes in software histories.

1.1 Semantic History Slicing

Semantic history slicing identifies a set of commits in a software history that relate to each other based on a certain criterion. For example, CSlicer [30,33] identifies and extracts a set of functionally-related commits that correspond to a specific high-level functionality; this set of commits forms a *history slice*. In CSlicer, the functionality is defined with a test suite, i.e., if all tests in the test suite pass, then functionality is available and works correctly. If a history slice contains only commits when the functionality is available and works, then the history slice is considered *valid*. *Git-bisect* [14] and *delta debugging* [53] isolate failure-inducing changes in version histories using a *divide-and-conquer-style* refinement approach, where a set of commits is partitioned and tested separately until a minimal subset that exposes the test failures is found.

The biggest challenge for precisely solving the semantic slicing problem lies in the very large number of possible (invalid) history slices for a given software history, i.e., we can create one history slice by choosing any subset of commits from the software history.

Existing solutions approach this problem from two different angles. CSlicer analyzes the latest program version to collect test coverage information and then computes an over-approximated set of commits that include changes to the covered elements. CSlicer trades accuracy for efficiency: it executes the test suite only once, but it conservatively assumes that all changes traversed by the test execution can potentially alter the test results. This assumption

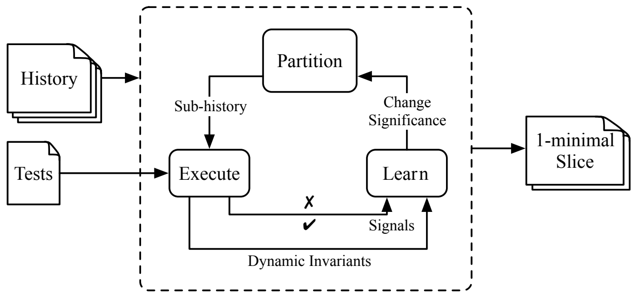


Fig. 1 Dynamic delta refinement loop.

results in potential inclusion of unnecessary or irrelevant commits into the history slice.

Divide-and-conquer-style techniques, such as delta debugging [53], guarantee accuracy of the result. Specifically, they guarantee that the resulting slice is *1-minimal* [53], i.e., it fails to satisfy the slicing criteria when any single commit is removed. Yet, these techniques can be very expensive to run as they execute the test suite multiple times, depending on the way history is partitioned and on the order in which partitions are tested.

1.2 Dynamic Delta Refinement

In this paper, we propose a precise semantic history slicing technique, DEFINER, based on *iterative refinement* and *change significance learning.* We discover relevance of changes to the target tests through successive test runs and utilize this information to guide the history partitioning and speed up the refinement process. We refer to this technique as *dynamic delta refinement.* Our key insight is that by comparing the runtime executions of two program versions – before and after a change – we can extract information about the precise impact of the changes at various program points. By combining impact information with test outcomes (pass or fail), we are able to accurately infer the significance of changes with respect to the target tests. In particular, if the tests still pass after removal of a change, then the removed change and its family of related changes are insignificant to the tested functionalities. We give more details on how such families of changes can be detected using dynamic program invariants generated by Daikon [10] in Section 4.

Figure 1 shows an overview of the delta refinement loop. Using the significance measurements of changes, dynamic delta refinement is able to efficiently find 1-minimal semantic history slices through well informed partition schemes. With much higher confidence, changes of less significance are removed first and, upon success, the analysis scope is reduced and the refinement continues recursively. The results of test executions, either successes or failures, are used to update significance ranking of the remaining changes. The ranking accuracy is improved with each execution, and the refinement loop terminates when the minimality condition is met. Note that the algorithm maintains a valid semantic slice throughout this process, so it can be interrupted at any time, which will return a valid best-effort result.

1.3 File-level Commit Splitting

Existing history slicing techniques operate at the *commit level*, i.e., a commit is either included in or excluded from the history slice. Using commits as a unit of granularity has the benefit of preserving the original commit structure and meta information, such as authors, change dates, and log messages. Yet, when a single commit includes changes related to different functionalities, commit-level treatment may result in adding unnecessary changes to the slice.

To increase the precision of history slicing, we introduce a *file-level commit splitting* operator, which splits a commit from the original version history into smaller units, called *file-level commits*, each consisting of changes to a single file. We choose to perform the split at the file level because files are natural units of change supported by the mainstream, language-agnostic SCMs such as Git (see Sections 5 and 6.3 for more details). By applying the split operator on the original software history, we create a *file-level software history*, which is used as the input to history slicing. File-level software history leads to a higher accuracy of slicing – the produced slice includes only commits that modify files relevant to the functionality of interest – without any substantial degradation in efficiency of the slicing tool.

1.4 Contributions

An earlier version of this work appeared in [32]. This paper expands the presentation of the technique, cleans up its formal underpinnings and provides a deeper comparison with related work. We have also significantly expanded the experimental evaluation, including a larger benchmark suite (20 cases vs. 8 used in [32]) and providing insights on the observed results. In addition, the idea of file-level splitting is completely new (the technique is presented in Section 5, its implementation – in Section 6.3, and evaluation of its effectiveness – in Section 7.5).

Overall, this paper makes the following contributions:

- We show how dynamic delta refinement can learn significance of changes with respect to a specific high-level functionality.
- We define a file-level commit splitting operator which splits the original software history into a set of file-level commits.
- We report on an implementation of a fully-automated precise semantic history slicing technique that leverages dynamic delta refinement and file-level commit splitting and operates on Java projects hosted in Git repositories.
- We compare our technique with prior work on history slicing in terms of precision and efficiency.

- We measure the effectiveness of dynamic delta refinement, comparing it to the basic partition scheme used by delta debugging.
- We measure the effectiveness of using the file-level commit splitting operator for further improving slicing accuracy.

1.5 Organization

The rest of this paper is structured as follows. Section 2 illustrates how dynamic invariants are used for learning change significance and guiding history partition, and how file-level commit splitting helps improve the slicing accuracy. Section 3 provides the necessary formal background for the rest of the paper. Section 4 formalizes the delta refinement algorithm for finding minimal semantic slices and proves its correctness. Section 5 describes file-level commit splitting and introduces a file-level history slicing algorithm. Sections 6 and 7 describe our implementation and evaluation, respectively. We compare our work with related approaches in Section 8 and conclude the paper in Section 9.

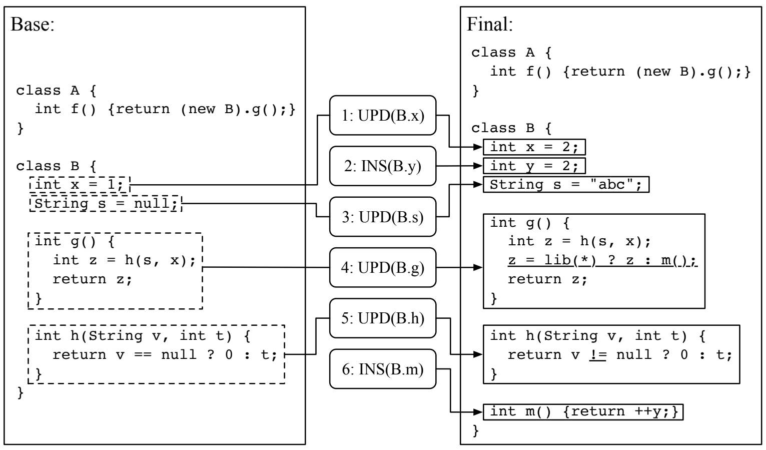
2 Examples

In this section, we use several examples to introduce necessary terminology, illustrate the dynamic delta refinement approach, and show the way we improve accuracy via file-level commit splitting.

2.1 Changes and Dependencies

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

The `lib(*)` method invoked by the `g()` method represents an external library invocation whose returned value is only known at runtime. We assume that the library method behaves deterministically but that its return value cannot be determined without executing it. The desired functionality of the program is captured by a unit test for `A.java` which asserts that the returned value of the method `A.f()` should be equal to 3 (see Figure 2b). We denote



Base:

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

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

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

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

Final:

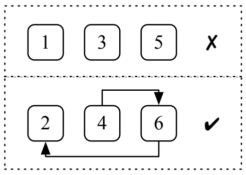
```
class A {
  int f() {return (new B).g();}
}

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

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

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

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

(a) Atomic changes.

Test:

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

(b) Target test.



(c) Change dependencies.

```
class B {
  int x = 1;
  int y = 2;
  String s = null;

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

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

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

(d) Class `B` after slicing.

Fig. 2 Atomic changes between the “base” and “final” versions of `A.java`.

this test by T. Note that the test assertion holds in the final version of the program but fails in the base version.

A semantic history slice is a subset of the changes which produces a well-formed and fully functional program that can still pass the test. Since we only care about a subset of the program behaviors captured by the test, some atomic changes are unnecessary. In our example, the minimal set of changes which qualifies as *a valid semantic slice* is $\{\boxed{2}, \boxed{4}, \boxed{6}\}$. The test T fails when any of these changes is missing and passes whenever all of them are present. The class `B` after applying the history slice with only three changes is shown in Figure 2d. Other changes are either never executed or do not alter the asserted values. Interestingly, the test passing property is not monotone, i.e., adding modifications may change the tests from passing to failing.

Atomic changes are not completely independent from each other. In order to construct a well-formed program, some changes have to be applied as prerequisites for others [30, 43]. For example, in Figure 2a, INS(`B.m`) depends on INS(`B.y`) since the method `B.m()` accesses the field `B.y` and requires the declaration of the field in order to compile; and UPD(`B.g`) depends on INS(`B.m`) since the new version of the method `B.g()` invokes `B.m()` (see Figure 2c). We call dependencies contributing to the well-formedness of programs *compilation dependencies*.

Since we are only interested in producing well-formed programs, the partition of changes has to obey the dependency relations. That is, reverting a subset of atomic changes results in a well-formed program only if the remaining changes have all their dependencies satisfied. The compilation dependencies can be computed statically, as we describe in Section 6.

2.2 Inferring Change Significance

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

In the example in Figure 2, the target test T passes in the final version. We can use this information to establish facts about the program variables at various program points by generating *dynamic invariants* [10] (denoted as I): likely invariants that may not generalize but that hold for the executed test. For simplicity, we refer to them as invariants from now on. For instance, `B::x == 2` is a field invariant which indicates that the value of the field `B.x` equals to 2 during the execution of tests on the final version of the program. Another example is `A.f()::return == 3` which is a method post-condition asserting that the return value of `A.f()` is 3.

We denote by H^- the set of reverted changes, I the set of invariants for the final version of the program, and I' the invariants after some changes are reverted. The row H^- of Figure 3 shows four possible cases of reverted changes for our example. The differences in the generated invariants before and after reverting changes are shown in row "$I \setminus I'$" of the table. The rows "$T(H^+)$" and "Signals" show the test outcomes and *significance signals* (which indicate the likelihood that a change impacts the test execution) learned for each case, respectively. We discuss each case in turn.

2.2.1 Case 1: Test Passing with Significance Update to Reverted Changes

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

Static analysis is unable to determine whether this change would affect the test outcome, because the return value of `lib(*)` is unknown. However, we are able to precisely detect the *impact* of reverting $\boxed{1}$ by comparing the new

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

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

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

Fig. 3 Change significance learning case by case.

set of invariants I' generated during the actual execution of the new program to the original invariants I. In this case, we observe that only one invariant disappears after reverting $\boxed{1}$, namely, `B::x == 2` (see row "$I \setminus I'$" in Figure 3). This indicates that the impact of reverting $\boxed{1}$ is local to the change itself and does not flow into other program points.

We assume that `lib(*)` returns `false` at runtime and thus the change on `B.x` does not propagate through – the returned value from `h(s,x)` is overwritten by `m()` which is independent of the change. The test outcome is unchanged and therefore, the value of `B.x` is considered to be insignificant. The significance scores of all changes are initialized to zero and in light of the test outcome, we decrease the score of $\boxed{1}$ by a predetermined constant (denoted by $\downarrow$ in Figure 3, row "Signals").

2.2.2 Case 2: Test Passing with Significance Update to Additional Changes

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

Since the test passes again, none of the three invariants in $I \setminus I'$ are consequential for the target functionality. This includes `B.h(I,S)::return == 0`, which implies that the return value of the function `B.h(I,S)` is likely not affecting the test result. Apart from $\boxed{1}$ and $\boxed{3}$ which are obviously insignificant (thus, we lower their significance scores), we could also infer that $\boxed{5}$ is insignificant. This is achieved by discovering, through static change impact analysis, the fact that $\boxed{5}$ could only possibly affect the return value of the function `B.h(I,S)` which is already shown insignificant to the test results. At this point, we have determined that the change set $\{\boxed{1}, \boxed{3}, \boxed{5}\}$ is insignificant for the target test. This information can be used to speed up the dynamic discovery of semantic history slices by prioritizing changes to revert in the next iteration ($\boxed{5}$ in our example).

2.2.3 Case 3: Test Failing by Determined Causes

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

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

2.2.4 Case 4: Test Failing by Undetermined Causes

When $H^- = \{\boxed{3}, \boxed{4}\}$ is reverted, the test fails but we cannot infer useful significance information. In this case, the test fails after reverting multiple atomic changes, and thus the causes for the failure are undetermined. The actual cause can be any one in the reverted changes or their combination. We equally increase their significance scores assuming each atomic change has the same probability being the actual cause of failure.

2.3 History Partition by Significance Ranking

The basic idea of history partition is inspired by delta debugging [53]. In the first iteration, the history is split into two halves which are then tested individually. If one of the partitions passes the test, then the process continues recursively on the successful partition. Otherwise, fine-grained partitions are produced by reverting fewer changes at once. For example, we can split the history into four similar-size change sets and revert each of them, one at a time. If none of the attempts are successful, then finer-grained partitions are produced until we reach a point where only a single atomic change is reverted at a time. Then we are able to classify the change precisely according to the test results. The process terminates when a 1-minimal history slice if found.

In this paper, we make two enhancements to the basic partition scheme: (1) before attempting basic partitions, we prioritize removal of low significance changes whenever possible, and (2) by precisely analyzing dependencies between changes, we detect compilation errors without the need to compile the program.

We use an example with a slightly more complex change history to illustrate our enhanced history partition scheme. In this example, there are eight atomic changes $\{\boxed{1}, \ldots, \boxed{8}\}$, adding two non-essential changes $\boxed{7}$ and $\boxed{8}$ on top of the

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

Fig. 4 Enhanced history partition scheme.

history in Figure 2. The set of essential changes is still $\{\boxed{2}, \boxed{4}, \boxed{6}\}$. Details of the additional changes can be found at `https://bitbucket.org/liyistc/gitslice/wiki/partition-example`.

The actual steps taken when analyzing this example are shown in Figure 4. Column "Partition" shows how some changes are reverted (in dashed boxes) and the others are kept (in solid boxes) in each round of partition. Columns "T" and "Signals" show the test results and corresponding significance signals learned (using methods illustrated in Section 2.2), respectively. During the first step ($n = 1$), the history is partitioned into two equal halves, i.e., $\{\boxed{1}, \boxed{2}, \boxed{3}, \boxed{4}\}$ and $\{\boxed{5}, \boxed{6}, \boxed{7}, \boxed{8}\}$. We keep one set and revert the other but only to find that the dependencies $\boxed{6} \rightarrow \boxed{2}$ and $\boxed{4} \rightarrow \boxed{6}$ are violated. In Figure 4, change dependency violations are represented with a "-" in column "T". No test run is needed so far.

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

During the third step ($n = 3$), we revert $\{\boxed{3}, \boxed{5}\}$ as suggested by their significance measurements and successfully reduce the scope down to only four atomic changes.

Similarly to the first step, neither half of the partition produced during Step 4 ($n = 4$) is a valid semantic slice. Therefore, we increase the partition granularity again in Step 5 ($n = 5$), reverting a single change at a time. This time, $\boxed{1}$ can be reverted which leaves a valid 1-minimal semantic history slice $\{\boxed{2}, \boxed{4}, \boxed{6}\}$. During the final step ($n = 6$), the delta refinement loop terminates because none of the changes can be successfully reverted.

For this example, six test runs are needed for finding the minimal solution using the enhanced partition scheme. In contrast, the basic partition scheme without significance learning or change dependency analysis [53], requires thirteen test runs and twelve additional (failed) compilations.

Although the examples in Figure 2 and Figure 4 illustrate the history slicing technique that operates directly on atomic changes, the same technique can also be applied on commits. We implemented and evaluated a tool that supports the latter (more details in Section 6.2).

2.4 File-level Commit Splitting to Increase Precision

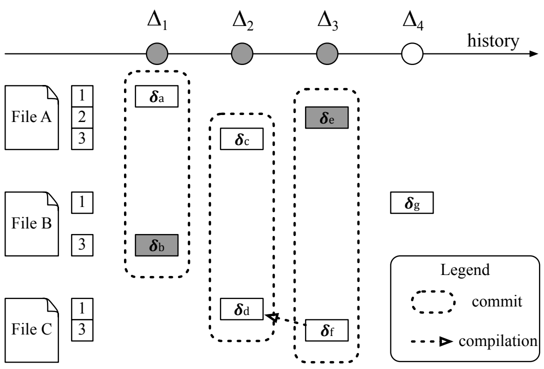


Fig. 5 An illustration of sources of imprecision in commit-level history slicing.

Figure 5 shows a diagram illustrating sources of imprecision in commit-level history slicing. The history segment H contains four commits, i.e., $H = \langle \Delta_1, \Delta_2, \Delta_3, \Delta_4 \rangle$. Each commit can be further broken into a set of hunks potentially spreading over multiple files. A *hunk* [11, 30] is a group of adjacent or nearby line insertions or deletions. For instance, Δ_1 has two hunks, δ_a and δ_b, over files A and B, respectively.

The only functionality-related changes in this example are δ_b and δ_e, shaded in gray. However, when performing history slicing on the commit level, we inevitably have to include unnecessary changes due to *commit dependencies* (two hunks within the same commit are *commit-dependent* on each other). For example, δ_a has to be included in the history slice because of δ_b, and δ_f is included because of δ_e (commit bundles are depicted as dashed boxes in Figure 5).

Unnecessary changes introduced by commit dependencies can induce further imprecision. For example, there is a compilation dependency between δ_f

and an earlier hunk δ_d (shown as a dashed arrow in Figure 5). The inclusion of δ_f, therefore, forces us to include δ_d as well. Thus, with commit-level history slicing, the best achievable result is a sub-history containing three commits: $\langle \varDelta_1, \varDelta_2, \varDelta_3 \rangle$.

In contrast, if instead of considering each commit as a whole, we treat each file-level hunk individually, then we are no longer constrained by commit dependencies. In fact, finer-grained history slicing at the file-level hunks allows us to reduce the number of unnecessary changes, resulting in only two hunks: δ_b and δ_e.

3 Preliminaries

In this section, we provide background and definitions for the rest of the paper.

3.1 Programs

To keep the presentation of algorithms concise, we step back from the complexities of the full Java language and concentrate on its core object-oriented features. We adopt a simple functional subset of Java from *Featherweight Java* [21], denoting it by P.

3.1.1 Language Syntax

The syntax rules of the language P are given in Figure 6. 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 [26].

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

Fig. 6 Syntax rules of the P language [21].

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

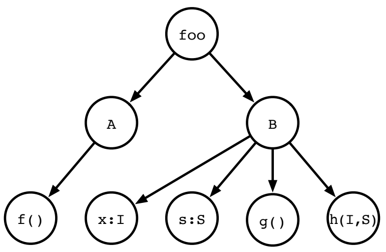


Fig. 7 AST of `A.java` at the base version.

class declaration has *members* including *fields* (i.e., $\overline{C\ f}$), *methods* (i.e., $\overline{M}$) and a single *constructor* (i.e., K). A method *body* consists of a single *return* statement; the returned expression can be a variable, a field access, a method invocation, an instance creation or a type cast.

3.1.2 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 $\textsc{Parent}(x)$. Figure 7 shows an AST for the base version of the program `A.java` from Figure 2a.

3.2 Program Semantics

Behavioral semantics of programs can be effectively captured by test executions.

3.2.1 Tests and Test Suites

We assume that semantic functionalities can be captured by tests and the execution trace of a test is deterministic [44]. For simplicity, a test can be abstracted into two parts – the setup code which initializes the testing environment and executes the target functionalities using specific inputs, as well as the oracle checks which verify that the produced results match with the expected ones. A test execution *succeeds* if all checks pass.

Definition 1 (Test). A *test* t is a predicate $t : P \mapsto \mathbb{B}$ such that for a given program p, $t(p)$ is true if the test succeeds, and false otherwise.

Definition 2 (Test Suite). A *test suite* is a collection of tests that can exercise and demonstrate the functionality of interest. Let test suite T be a set of test cases $\{t_i\}$. We write $p \models T$ if and only if program p passes all tests in T, i.e., $\forall t \in T \cdot t(p)$.

3.2.2 Dynamic Invariants

Dynamic invariants [42] are likely invariants that are discovered from program executions. They assert predicates that hold during the execution at specific program points including procedure entries and exits, and aggregate program points of multiple class instances. We are particularly interested in three types of predicates:

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

A wide range of dynamic invariants is detected and reported by Daikon [10], but only a subset of the invariants are used in this paper. In particular, we consider a subset of the invariants which involve a single program variable, including comparisons with constants (e.g., `x == K`, `x == K1 (mod K2)`, `K1 <= x <= K2`, `x != null`), single-valuedness (e.g., `x has only one value`), and value range (e.g., `x one of {a,b}`).

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

3.3 Changes and Change Histories

Based on the program representations used, either structured ASTs or unstructured plain texts, we define the AST- and Line-based views of changes and change histories.

3.3.1 AST-Based View

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

Definition 3 (Atomic Change). An *atomic change* operation $\delta : \Gamma \nrightarrow \Gamma$ is a partial function which transforms $r \in \Gamma$ producing a new AST r' such that $r' = \delta(r)$. An atomic change operation can be either an *insert*, *delete* or *update* (see Figure 8). An insertion $\textsc{Ins}((x, n, v), y)$ inserts a node x with an identifier n and a value v as a child of a node y. A deletion $\textsc{Del}(x)$ removes a node x from the AST. An update $\textsc{Upd}(x, v)$ replaces the value of a node x with v.

A change operation is *applicable* on an AST if its preconditions are met. For example, an insertion $\textsc{Ins}((x, n, v), y)$ is applicable on r if and only if $y \in V(r)$. Insertion of an existing node is treated the same as an update.

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

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

Fig. 8 Types of atomic changes [12]. Refer to Section 3.1.2 for AST-related symbols.

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

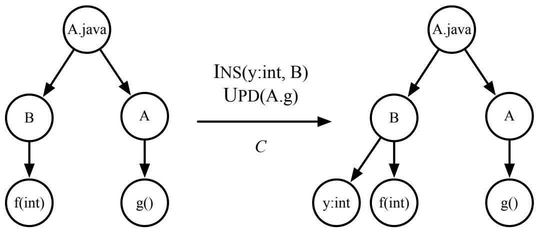


Fig. 9 Visualizing a change set C as a sequence of atomic changes applied on ASTs.

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 [5]. For instance, in Figure 9, the change set C consists of an insertion of a new node `y` under `B`, followed by an update of the node `g`.

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

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

We write $H' \subseteq H$ 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 similarly to that of change sets. We use $SH(H)$ to denote the set of all sub-histories of H.

3.3.2 Line-Based View

SCM tools represent changes using the *line-based view*. The smallest unit for line-based changes is a hunk.

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

For example, Figure 10 shows a hunk of one line deletion and two line insertions, marked by "-" and "+", respectively. The *context* (the lines not marked by "-" or "+" in Figure 10) that comes with a hunk is useful for ensuring that the hunk can be applied at the correct location even when the line numbers change for the target program texts.

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

Fig. 10 Line-based view of changes represented as a hunk.

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

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

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

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

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

4 Definer: Dynamic Delta Refinement Algorithm

In this section, we give a detailed presentation of Definer– the *dynamic delta refinement algorithm* for precise semantic history slicing.

4.1 Precise Semantics-preserving Slice

Consider a program $p_0 \in P$ and its k subsequent versions $p_1, \ldots, p_k$ such that each p_i is well-formed.

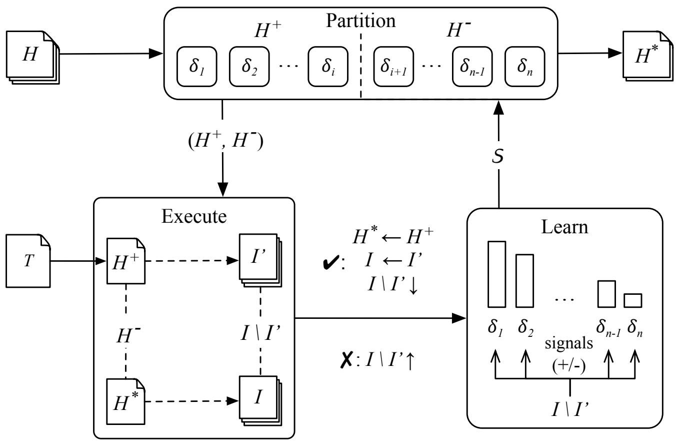

H
Partition
H^+
H^-
δ_1 δ_2 ⋯ δ_i δ_{i+1} ⋯ δ_{n-1} δ_n
H^*
(H^+, H^-)
S
Execute
T
H^+
I'
H^-
$I \setminus I'$
H^*
I
✔: $H^* \leftarrow H^+$ $I \leftarrow I'$ $I \setminus I' \downarrow$
✗: $I \setminus I' \uparrow$
Learn
δ_1 δ_2 … δ_{n-1} δ_n
signals (+/-)
$I \setminus I'$

Fig. 11 Dynamic delta refinement overview.

Definition 10 (Semantics-preserving slice [33]). Let H be the original 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, denoted by $H^* \subseteq_T H$, is a sub-history $H' \subseteq H$ such that $H'(p_0) \models T$.

Of course, H is a semantics-preserving slice of 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 [33]: finding it requires $2^{|H|} - 1$ tests in general.

Therefore, we aim at computing an approximation of the optimal solution which still has good practical precision guarantees. We say that a sub-history H^* of H is a *1-minimal semantic slice* if H^* is semantics-preserving, and reverting any single change in H^* would break the semantic properties.

Definition 11 (1-Minimal Semantic Slice). Let H^* be semantics-preserving, i.e., $H^* \subseteq_T H$. H^* is a *1-minimal semantic slice* of H if $\forall \delta \in H^* \cdot (H^* \setminus \{\delta\}) \not\models T$.

4.2 Algorithm Description

Given a history H and a test suite T, to compute a 1-minimal semantic slice H^*, our algorithm iteratively goes through three phases: *partition*, *execution* and *learning*, as shown in Figure 11. To implement each phase, the delta refinement algorithm maintains three data structures: (1) H^*, the current minimal semantics-preserving history slice, which is always an over-approximation of the 1-minimal solution and can be returned as a sub-optimal solution if the

Initialization:

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

Partition:

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

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

Execution and Learning:

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

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

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

Fig. 12 The dynamic delta refinement algorithm.

refinement process terminates prematurely; (2) I, the set of dynamic invariants generated from the last successful test execution and updated after every successful run; (3) $\mathcal{S} : \varDelta \to \mathbb{R}$, the change significance ranking – a function from atomic changes to real numbers, updated according to the outcomes from the execution phase. Multiple instantiations of the algorithm exist, depending on which strategies are used at the partition and learning phases. To keep the presentation general, the algorithm is shown in Figure 12 as a set of generic rules specifying the minimal requirements for each phase. To apply each rule, the conditions above the horizontal bar need to be satisfied. The expressions below the bar are the corresponding consequences after the rules take effects.

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

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

The *random partition* splits the current minimal semantic slice H^* into two non-empty sub-histories, H^+ and H^-, randomly, when the length of H^* is greater than one. Then H^+ is kept while H^- is reverted. The relative sizes of H^+ and H^- can be adjusted according to heuristics during the execution.

For example, a smaller H^+ can reduce a larger chunk of non-essential changes if the tests pass, but it usually has a lower chance of success assuming that essential changes are uniformly distributed. One effective heuristics we use is to gradually increase the size of H^+ when a test fails and decrease it otherwise.

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

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

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

When H^+ passes T, the Pass rule applies. We use the invariant deltas to match each affected variable and program point involved with atomic changes that might be the cause. This matching step is performed using a simple local static change impact analysis [1]. For each affected method postcondition, we collect all statements within the method body that have potential impacts on the method return (e.g., changed value flows into the return). For instance, using a simple backward data-flow analysis, the invariant "`B.g()::return == 3`" in Figure 3 is matched to $\boxed{4}$ which directly updates the returned variable `z`. Similarly, for each method precondition, we consider every call site and collect statements preceding the method invocation which potentially impacts the corresponding input parameters. Finally, for invariants on fields, we analyze all field access sites and perform a similar backward analysis. The significance of each matched change is decreased. We update H^* to H^+ and recursively apply partition rules on H^*.

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

Termination Condition. The algorithm should never attempt the same partition (H^+) twice and it terminates whenever H^* becomes empty or the 1-minimal condition (see Definition 11) is met – $\forall \delta \in H^* \cdot (H^* \setminus \{\delta\}) \not\models T$.

4.3 Soundness and Completeness

The following theorem states that the algorithm is sound.

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

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

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

Theorem 2 *(Completeness). Given a history H and a test suite T, the algorithm using fair partition schemes always terminates with finitely many rule applications.*

We give a proof sketch of the theorem. Suppose the algorithm does not terminate. Since H^* has finite number of changes initially and its length is monotonically decreasing, $|H^*|$ has to eventually stay constant. Because of the fairness condition, every atomic change in H^* is eventually reverted and tested. If none of the tests pass, then the 1-minimal condition is met. If one of the tests passes, then $|H^*|$ should decrease. Both cases lead to contradictions. Therefore, the algorithm always terminates.

5 History Slicing with File-Level Splitting

The history slicing techniques presented so far operate at the *commit level.* Using commits as the smallest units for doing history slicing has the benefit of preserving the original commit structure and traceability from the high level semantic property to its corresponding commit-level meta information such as authors, change dates, and log messages. This information can be useful in supporting other downstream maintenance tasks.

In practice, commits usually contain changes to several files, classes and methods, organized as hunks. Different hunks in the same commit are not necessarily logically related or relevant to the same functionality. Thus, considering a commit as an atomic unit does not allow us to remove many unnecessary changes for the target features.

As shown earlier in Section 2.4, granularity at which the history slicing technique is applied has a significant impact on the final results. In the rest of this section, we describe an optional enhancement to the history slicing technique presented in Section 4 with *file-level splitting of commits.* In essence, the enhancement is achieved by introducing a "split" operator which breaks a commit into a set of smaller commits, called *file-level commits*, each including changes to a single file. By splitting large commits carrying changes possibly unrelated to each other, we can effectively reduce hunk dependencies and achieve a much better precision in history slicing. The commit splitting is orthogonal to history slicing – existing history slicing techniques can be applied on file-level commits directly without any modification.

We now formalize the file-level split operator and use an example to demonstrate its effect on Definer. A split operator transforms a commit into a sequence of (possibly smaller) commits which have equivalent overall effect on programs. More formally,

Definition 12 (Split Operator). Let $\Delta = \{\delta_0, \ldots, \delta_n\}$ be a commit containing a set of hunks. SP is a *commit split operator* if $SP(\Delta) = \{\Delta_i, \ldots, \Delta_j\}$ is a partition of Δ, where $\Delta_i, \ldots, \Delta_j$ are the new split commits.

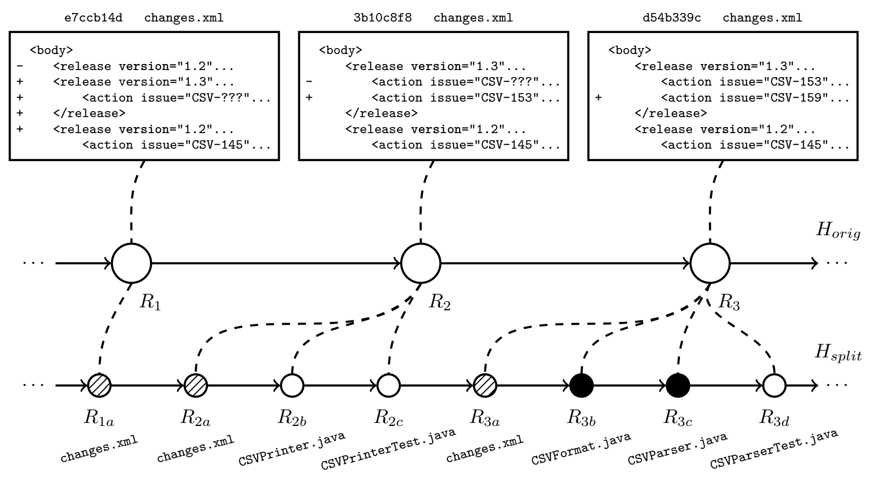
Trivially, we also have for any program p, $(\Delta_i \circ \cdots \circ \Delta_j)(p) = \Delta(p)$. The result of applying a split operator on a history is simply to apply it to each of the commits within the history: $SP(H) = \langle SP(\Delta_1), \ldots, SP(\Delta_k)\rangle$.

There is more than one way to implement a split operator. For instance, this definition does not impose any constraints on the size of the split. In theory, a history can be split into units as small as atomic changes. However, this is often too difficult to implement on top of language-agnostic text-based version control systems such as Git. The biggest challenge is that there may not be a one-to-one mapping between atomic changes (which is defined over ASTs) and hunks (which is the smallest unit in text-based version control tools). For example, a hunk can contain several atomic changes, and there is no easy way to split a hunk into smaller units without proper language-specific support. Notably, at the file-level, a set of atomic changes within the same file can be mapped to a set of hunks. Therefore, we argue that file-level commits are natural units of change which align well with language-agnostic version control systems and allow much easier integration with existing version control tools such as Git.

For the above-mentioned reasons, we propose a *file-level split operator* – SP_{file}, which partitions a commit according to the files modified in the commit. In other words, $SP_{file}(\Delta)$ is the set of equivalence classes induced by the equivalence relation: $\delta_i \sim \delta_j : \delta_i$ and δ_j changes the same file.

Figure 13 demonstrates Definer with file-level split on the Apache CSV project, which is a popular open-source project for processing CSV files. The feature identified by "CSV-159", first requested on October 14th, 2015, enables case insensitive matching of header names for CSV files. Figure 13 shows a fragment of the change histories for the CSV project.

If we run Definer on H_{orig} to perform history slicing for this feature, the resulting slice consists of three commits, R_1, R_2 and R_3, which are highlighted in Figure 13. In the last commit (R_3) of the slice, a developer made changes in four different files: `changes.xml`, `CSVFormat.java`, `CSVParser.java`, and


e7ccb14d changes.xml

```
  <body>
-     <release version="1.2"...
+     <release version="1.3"...
+         <action issue="CSV-???"...
+     </release>
+     <release version="1.2"...
          <action issue="CSV-145"...
```

3b10c8f8 changes.xml

```
  <body>
      <release version="1.3"...
-         <action issue="CSV-???"...
+         <action issue="CSV-153"...
      </release>
      <release version="1.2"...
          <action issue="CSV-145"...
```

d54b339c changes.xml

```
  <body>
      <release version="1.3"...
          <action issue="CSV-153"...
+         <action issue="CSV-159"...
      </release>
      <release version="1.2"...
          <action issue="CSV-145"...
```

Fig. 13 An example illustrates how splitting technique helps definer do more precise slicing.

`CSVParserTest.java`. One of these changes, to `changes.xml`, has a hunk dependency on a change introduced in the commit R_2, which itself has a hunk dependency on a change made in R_1. The details of the changes to the file `changes.xml` are shown in boxes (Figure 13).

In contrast, if we run Definer on the same feature of the split history H_{split}, the resulting slice only consists of two (smaller) commits. First, the file-level split operator produces a finer-grained software history, such that each commit contains changes only to a single file. Figure 13 illustrates this fine-grained history H_{split} with the smaller nodes representing the file-level commits. Second, we run Definer on H_{split}, which returns two commits, namely, R_{3b} and R_{3c}. These commits modify two files, (`CSVFormat.java` and `CSVParser.java`), which are required for the tests to pass. Interestingly, both R_{3b} and R_{3c} come from a single commit (R_3) in the original software history. Thus, the final history slice contains far fewer changes – one commit (R_3) instead of the three that would have been obtained by Definer without the file-level split. This also implies that only changes made in the commit R_3 are relevant to the feature CSV-159.

6 Implementation and Optimizations

In this section, we describe our implementation of a semantic slicing tool based on the dynamic delta refinement algorithm. Our tool, called Definer, targets Java projects hosted in Git repositories. We describe the details of the implementation and several optimizations that make our tool more practical.

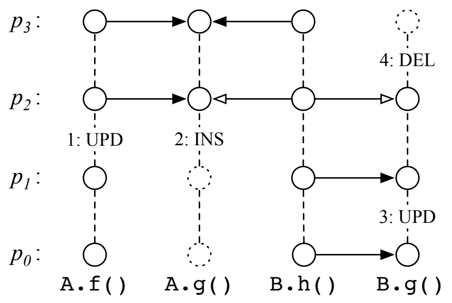


Fig. 14 Analyzing change dependencies.

6.1 Change Dependency Analysis

To avoid running into compilation errors, we perform a pre-analysis for each version in the software history and compute direct dependencies for all changed AST nodes. This analysis produces a *multi-version change dependency graph*, which we illustrate with an example in Figure 14.

In this example, there are four program versions, i.e., p_0, p_1, p_2 and p_3, all of which are well-formed (syntactically correct and compilable). There are three changed nodes, i.e., methods `A.f()` and `A.g()`, which belong to class `A`, as well as `B.g()`, which belongs to class `B`. There is also a node `B.h()` which stays unchanged. Class `B` is a sub-class of `A`. Each node has a separate time-line on which its changes are labeled. In particular, `A.f()` has an update between p_1 and p_2; `A.g()` is inserted between p_1 and p_2; and `B.g()` is updated after p_0 but deleted after p_2. In Figure 14, solid arrows represent *necessary* dependencies while empty arrows represent *sufficient* dependencies. For instance, a method invocation of `g()` in `B.h()` makes `B.h()` necessarily depend on `B.g()` before p_2. But when `g()` is introduced in the super-class `A` in version p_2, both definitions of `g()` are sufficient dependencies of `B.h()`, i.e., existence of either one of them would satisfy the compilation requirement due to method inheritance.

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

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

6.2 Git Adaptation

The generic algorithm discussed in Section 4 operates on the level of atomic changes. To work with Git, we treat the set of atomic changes belonging to the same commit as a bundled group. The partition algorithm is adjusted such that changes in the same group always stay together, and the significance score for a group is computed as the sum of the scores of its members. Apart from dependencies between atomic changes, we also analyze hunk dependencies between commits so that the history slices can be applied without causing a Git merge conflict. History partitions which do not comply with the hunk dependencies are discarded immediately without running any tests.

6.3 Implementing the Split Operator

Since we would like the slicing result to be compatible with Git, in practice, we rely on the *interactive staging* [17] feature of Git to implement the split operator. Interactive staging enables modifications of commits where individual hunks within a commit can be selected and regrouped to form new commits. It is also necessary to keep the mapping between the split and the original commits, so that the traceability of historical meta-data can be restored.

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

Require: $\Delta(p) = p'$
Ensure: $splits = SP_{file}(\Delta)$
 9: procedure SplitCommit($p, p', \Delta$)
10:     $splits \leftarrow \langle\rangle$
11:     git checkout $p'$                          ▷ Checkout program version $p'$.
12:     git reset $p$                        ▷ Unstage all changes in the commit $\Delta$.
13:     while there is unstaged file do
14:         git add -- $f$                                 ▷ Stage a single file "$f$".
15:         git commit -m "Origin: $\Delta$"      ▷ Commit $\Delta_f$ with log indicating origin.
16:         $splits \leftarrow splits, \Delta_f$                  ▷ Add $\Delta_f$ at the back of $splits$.
17:     end while
18:     return $splits$
19: end procedure
```

Fig. 15 An algorithm implementing file-level split operator for Git history.

Figure 15 outlines the process of splitting an original history H into an equivalent history H_{split} which contains only file-level commits (see the procedure SplitHistory). The procedure also returns a mapping m which indicates the origins of the split commits in H_{split}. The procedure iterates through all the input commits in order (Line 3 to 6) and calls a sub-routine SplitCommit.

The procedure SplitCommit splits a single commit into a sequence of file-level commits each only containing hunks within a single file. To process the target commit Δ, we first check out the version p' right after it (Line 11). Then we use "`git reset`" to unstage all changes in the commit Δ, which will move those changes out of the *staging area* [16] and allow them to be reorganized later (Line 12). The Git command "`git add --`" provides a way of staging changes to individual files (Line 14), allowing them to be committed separately (i.e., commit Δ_f at Line 15). Finally, the sequence of file-level commit is returned (Line 18).

For instance, when applying SplitCommit on the commit R_3 (`d54b339c`) in our example, the repository is first checked out to the version `d54b339c`. Then the changes made in `d54b339c` get unstaged through the command `git reset 3b10c8f8`, resulting in changes over four different files. The changes for each file get added to the stagging area and committed individually using the command `git add -- changes.xml`, etc.

6.4 Tooling

Definer is written in Java and yields fully-automated analysis of projects built with Maven [38]. We use JGit [23], a Java implementation of Git, for repository manipulation and commit-level hunk dependency analysis [30]. We use a modified version of ChangeDistiller [13] for extracting AST-level atomic changes from Git commits. We also use the Apache Byte Code Engineering Library (BCEL) [2] to analyze dependencies among atomic changes, Daikon [10] for dynamic invariant detection, and Soot [49] for performing local change impact analysis.

We created a baseline version Definer-Default which operates on the original commits with all optimizations enabled, and an enhanced version Definer-Split which has the exact same configuration but operates on the file-level commits. To evaluate the effectiveness of various optimizations, we also created Definer-Learn which disables compilation failure detection based on change dependency analysis and Definer-Basic which also disables significance learning. The source code of Definer is available online at the following URL: `https://bitbucket.org/liyistc/gitslice`.

7 Evaluation

We evaluate Definer w.r.t. both its precision and performance using a benchmark suite obtained from open source software repositories. The goal of our empirical evaluation is to answer the following research questions:

Table 1 Statistics of tested software projects.

Projects	#Files	LOC
`compress`	307	37,768
`io`	233	29,188
`lang`	326	74,479
`net`	270	27,845
`csv`	30	5,538

RQ1: How does the precision of history slices produced by Definer compare to those produced by CSlicer?

RQ2: How effective are the change significance ranking and change dependency analysis for guiding history partitions when compared with the basic partition scheme used by delta debugging?

RQ3: How do different partition schemes affect the performance of Definer?

RQ4: How effective is the file-level commit splitting for improving the precision of Definer?

7.1 Subjects

We evaluated Definer on a benchmark consisting of 20 target functionalities randomly selected from the DoSC dataset [55]. DoSC consists of 81 target functionalities collected from repositories on GitHub.

Each item in DoSC is a high-level functionality identified with a unique key, which refers to its corresponding issue key on the JIRA issue tracker [25]. Each functionality is accompanied by a test suite for it, where the code of functionality and the test cases are committed together. DoSC labels the starting version and ending version of the software history which determines the entire life cycle of the development of a functionality.

In order to test the history slicing capabilities of Definer, all the experiments require an original history segment (H) and a target test suite (T) designated for certain high-level functionality. In DoSC, this information is recorded in the YAML format [51], which is programmatically extractable and can be directly fed into Definer. Thus, we ran Definer on the 20 selected functionalities and collected the experimental data to answer our research questions.

The selected functionalities were originally developed in five open source projects, namely, Apache Commons Compress Library (`compress`) [6], Apache Commons IO Library (`io`) [22], Apache Commons Lang Library (`lang`) [28], Apache Commons Net Library (`net`) [39], and Apache Commons CSV Library (`csv`) [7]. These projects are all written in Java and their software histories are freely accessible online.

The selected projects are under active development and their sizes range from 5 to 75 KLOC. Statistics about each project is shown in Table 1. Columns

Table 2 Functionalities details and descriptions.

Projects	ID	Issue Key	Functionality Descriptions	End Version	$\|H\|$	$\|T\|$	$\|H^*\|$
compress	C1	COMPRESS-327	Support in-memory processing for ZipFile	b29395d	148	18	26
	C2	COMPRESS-369	Allow archiver extensions through a standard JRE ServiceLoader	b29395d	148	2	11
	C3	COMPRESS-373	Support writing the "old" lzma format	b29395d	148	1	14
	C4	COMPRESS-374	Add support for writing lzma streams in 7z	b29395d	148	8	16
	C5	COMPRESS-375	Allow the clients of ParallelScatterZipCreator to provide ZipArchiveEntryRequestSupplier	b29395d	148	2	1
io	I1	IO-173	FileUtils.listFiles() doesn't return directories	b1b9f1a	136	2	16
	I2	IO-275	Add option to ignore line endings	b1b9f1a	136	2	1
	I3	IO-288	Supply a ReversedLinesFileReader	b1b9f1a	136	18	2
	I4	IO-290	Add read/readFully methods to IOUtils	b1b9f1a	136	2	5
	I5	IO-305	New copy() method in IOUtils that takes additional offset, length and buffersize arguments	b1b9f1a	136	10	26
lang	L1	LANG-883	Add StringUtils.containsAny(CharSequence, CharSequence...) method	76cc69c	262	1	TO
	L2	LANG-993	Add zero copy write method to StrBuilder	76cc69c	262	10	6
	L3	LANG-1006	Add wrap (with String or char) to StringUtils	76cc69c	262	2	14
	L4	LANG-1080	Add NoClassNameToStringStyle implementation of ToStringStyle	76cc69c	262	8	TO
	L5	LANG-1093	Add ClassUtils.getAbbreviatedName	76cc69c	262	2	TO
net	N1	NET-525	Added missing set methods on NTP class and interface	abd6711	269	14	14
	N2	NET-527	Add SimpleNTPServer as example and for testing	abd6711	269	1	20
csv	S1	CSV-159	Add IgnoreCase option for accessing header names	7310e5c	79	1	3
	S2	CSV-175	Support for ignoring trailing delimiter	7310e5c	79	11	14
	S3	CSV-180	Add withHeader(Class<? extends Enum>) to CSVFormat	7310e5c	79	2	16

Table 3 Average execution time: CSLICER vs. DEFINER-DEFAULT vs. DEFINER-SPLIT.

Mode	Time (s)
CSLICER	26
DEFINER-DEFAULT	754
DEFINER-SPLIT	1,556

"#Files" and "LOC" show the number of Java files and the total lines of code, respectively; these numbers are for the latest version.

The details about each functionality are given in Table 2. Column "ID" lists subject identifiers. Column "Issue Key" lists the corresponding issue key of the functionality on JIRA. Column "End Version" shows the target commits which correspond to the final version in our history segment. Columns "$|H|$" and "$|T|$" show the length of the original history segments and the sizes of the target test suites, respectively. Column "$|H^*|$" shows the length of the manually verified 1-minimal history slice.

For all of the experiments, we set a two-hour time limit for running DEFINER on each functionality. On three examples, L1, L4, and L5, DEFINER did not terminate within this time limit on any configuration for any experiment. We excluded these from further consideration.

All the experiments were conducted on a 4-core Intel i7-6700 CPU @ 3.40GHz machine with 16GB of RAM, running Ubuntu 17.04.

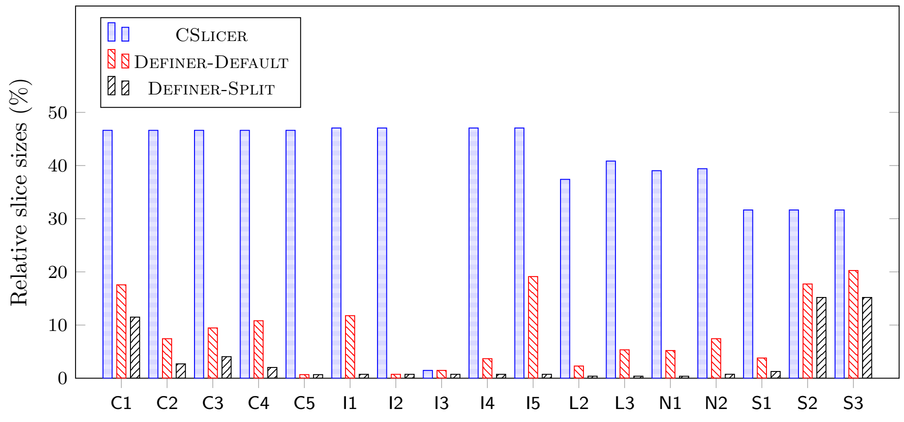


Fig. 16 Lengths of slices: CSlicer vs. Definer-Default vs. Definer-Split.

7.2 RQ1: How Does the Precision of History Slices Produced by Definer Compare to Those Produced by CSlicer?

The first experiment aims to compare Definer with CSlicer in terms of the precision of the produced history slices. We use the default configuration of Definer (Definer-Default) which adopts a simple partition scheme that first reverts commits with negative significance scores. The relative history slice length for each subject is computed as the length of the produced history slice divided by the original history length, i.e., $|H^*|/|H|$. The results are shown in Figure 16. As an example, consider the subject I1. The length of the history slice computed by Definer-Default is 11.76% of the original history, while the length of the history slice computed by CSlicer is 47.06% of the original history.

In fact, the history slices found by Definer-Default are always shorter or equal to those computed by CSlicer (79% shorter on average). Furthermore, all history slices produced by Definer-Default are manually verified to be 1-minimal while CSlicer does not guarantee minimality. For example, out of 148 commits from the subject C1, Definer-Default finds a slice of length 26 which is shorter than the one of length 69 reported by CSlicer.

Table 3 shows the comparison of execution time of CSlicer and Definer-Default (we discuss the last row later in the document). On average, CSlicer took 26s to obtain the history slice, while Definer-Default took 754s.

In summary, Definer substantially outperforms CSlicer in terms of precision: history slices obtained by Definer-Default are 79% shorter on average than history slices obtained by CSlicer. Although Definer-Default takes more time, on average, we consider this performance overhead to be reasonable, because history slicing is often performed as an off-line maintenance task [33, 34].

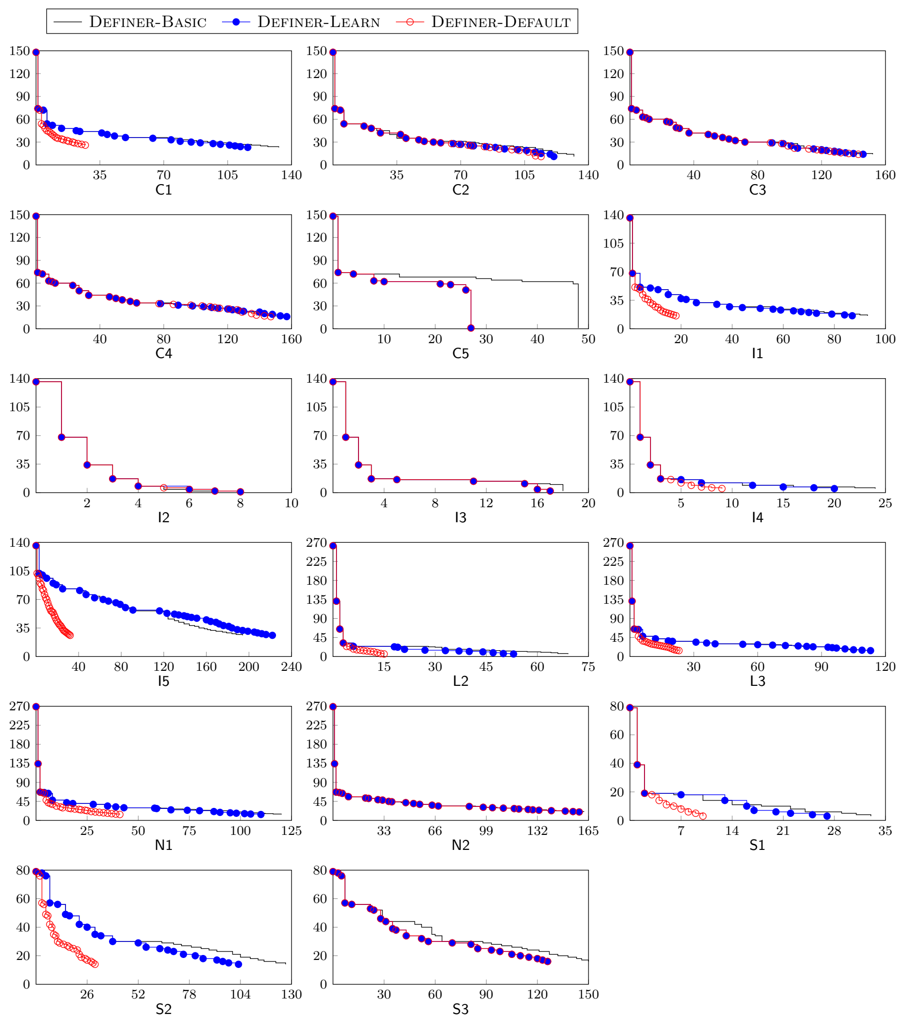


Fig. 17 History reduction per test run.

7.3 RQ2: How Effective are the Change Significance Ranking and Change Dependency Analyses for Guiding History Partitions when Compared with the Basic Partition Scheme Used by Delta Debugging?

The second experiment evaluates the effectiveness of using change significance ranking and change dependency analyses in speeding up the delta refinement loop. We compared three configurations of DEFINER in terms of the number of test runs needed to find 1-minimal history slices: (1) DEFINER-DEFAULT as described earlier, (2) DEFINER-LEARN which disables compilation failure detection based on change dependency analyses, and (3) DEFINER-BASIC which also disables significance learning and thus is effectively equivalent to delta debugging which applies the basic (random) partition scheme.[1] All configurations still apply hunk dependency analyses which detect Git merge failures without actually applying the commits.

The results of the comparisons are shown in Figure 17 where the length of H^* (y-axis) is plotted as a function of the number of test runs (x-axis). In general, DEFINER-DEFAULT and DEFINER-LEARN require fewer test runs than DEFINER-BASIC to reach the minimal solution. In most of the cases, the advantage of significance learning is obvious, especially for cases such as C5, L2 and S2 where DEFINER-LEARN requires on average only about 71.39% of test runs compared with DEFINER-BASIC. In addition, change dependency analyses which prevent test runs on non-compilable programs helped extend this advantage further – it only takes about 33.87% of test runs.

There are some cases where DEFINER-DEFAULT and DEFINER-LEARN show no (or small) advantage over the DEFINER-BASIC configuration, such as C2, C3, C4, and N2. By inspecting these cases, we determined that they were caused by non-Java commits, i.e., commits that change only non-Java files, e.g., XML files, but are important for the success of building/testing of the functionality. As an example, there is a commit (`4379a681`) in the original history of N2, which only modifies `pom.xml` to fix an incorrect XML snippet. Without this commit, the project cannot be built, so this commit must be kept in the history slice. However, DEFINER only computes significance scores for atomic changes defined on Java code entities. As a result, for this particular example, even though DEFINER discovers that dropping this commit will lead to build failure, this information does not help in guiding the subsequent partitions. Even worse, any commit dropped together with this commit will be ranked higher by DEFINER due to the fact that dropping them fails the build, which means DEFINER learns false signals in such scenarios.

In summary, we find that change significance ranking may lead to substantial savings in the number of executed tests. For those subjects where the change significance ranking does not help, this happens because our approach does not compute significance scores of changes for non-Java files; we plan to address this in our future work.

7.4 RQ3: How Do Different Partition Schemes Affect the Performance of DEFINER?

We also experimented with three different partition schemes, namely, NEG, NONPOS, LOW-3, and their combination, COMBINED (COMBINED is used in the previous experiments). All schemes follow the general steps described in Section 2.3 with different partition priorities at the beginning of each iteration. The NEG scheme only reverts commits which have negative scores. It is the most conservative one among the three. NONPOS is the most aggressive one which reverts all commits with non-positive scores. LOW-3 always reverts the

[1] We could not directly compare with the original implementation in [52], which does not work with Git repositories.

Table 4 Comparisons of different partition schemes in terms of the number of test runs.

ID	Neg	NonPos	Low-3	Combined
C1	116	116	116	**27**
C2	122	121	122	**114**
C3	**143**	146	**143**	**143**
C4	156	157	156	**147**
C5	**27**	**27**	**27**	**27**
I1	86	87	86	**18**
I2	**8**	**8**	9	**8**
I3	**17**	**17**	19	**17**
I4	20	20	19	**9**
I5	206	222	206	**32**
L2	52	53	52	**15**
L3	109	113	109	**23**
N1	109	110	109	**41**
N2	**159**	**159**	**159**	**159**
S1	27	27	23	**10**
S2	101	103	101	**30**
S3	**125**	126	**125**	126

lowest one third of the commits according to their significance ranking. Using Combined, Definer attempts all three partitions at each partition phase according to the three different schemes, in the order of first Neg, then NonPos, and finally Low-3. If any one of the partitions succeeds at the execution phase, we move on to the next iteration.

The results of this experiment are shown in Table 4, where each column lists the number of test runs required to reach the minimal solution and the best configurations for each row are in bold. All four partition schemes perform well on at least one subject. For example, Low-3 required the smallest number of test runs for C3, C5, N2, and S3. All partition schemes perform equally well on two examples, namely, C5 and N2. These correspond to the cases where change significance ranking is not as effective (see Section 7.3).

In summary, the choice of a partition scheme can impact the performance of Definer. We find that each scheme performs the best on at least one subject. This opens an interesting research direction: predicting what scheme to use for a given subject. Meanwhile, we suggest that the default configuration should use the Combined strategy, which achieved the best performance in 16 out of 17 examples.

7.5 RQ4: How Effective is the File-level Commit Splitting for Improving the Precision of Definer?

We performed an experiment comparing Definer-Default with Definer-Split. We compared these two approaches in terms of the precision of their outcomes and their execution time. Figure 16 illustrates the effectiveness of the file-level splitting operator for reducing the length of the resulting history slices. The history slices computed by Definer-Split are shorter than those done by Definer-Default on most of subjects (15 out of 17 subjects). On two out of 17 subjects Definer-Default and Definer-Split obtained the same length of history slices. On average, the results of Definer-Split are 60% shorter than those of Definer-Default.

Regarding the execution time, as shown in Table 3, on average, Definer-Split took 1,556s to finish. Recall that Definer-Default took, on average, 754s. The extra cost of Definer-Split is likely acceptable for most users considering that the average history length reduction is as high as 60%.

In summary, file-level commit splitting is an effective approach for improving precision of Definer. If a user can afford extra execution time and loss of information from the original software history (Section 6.3), file-level commit splitting should be enabled by default.

7.6 Summary

To summarize, we evaluated the precision and performance of Definer empirically on a benchmark set of real-world software projects. We demonstrated that Definer produces more precise history slices than existing state-of-the-art techniques such as CSlicer, although Definer exhibits longer execution times. We consider the performance losses acceptable for the achieved precision gain. Moreover, in the majority of cases, Definer outperforms the basic partition scheme used by delta debugging, which is attributed to the change significance ranking learned during the refinement process. With the Combined partition scheme applied, Definer achieves precise slicing results in an efficient manner. Hunk dependencies significantly affect the precision of Definer. By performing file-level splitting of the history and running Definer on the resulting split history, the precision of history slicing gets improved further.

7.7 Threats to Validity

Our experiments are subject to common threats to validity.

7.7.1 External

Subjects used in our evaluation may not be representative. To mitigate this threat, we used subjects used in prior research on history slicing; these subjects are taken from large open-source projects covering diverse domains.

We ran all the experiments on a single machine, and our findings related to execution time might differ on another machine. During the implementation of our tool, we have used several machines and observed similar trends on these machines.

7.7.2 Internal

Our implementation of Definer (and all of its variants), as well as our scripts for running the experiments, may contain bugs. Two of the authors worked closely on the implementation of the tool and frequently reviewed code together. We have also checked for various outliers in our results that indeed discovered a couple of bugs, which have been fixed since then.

7.7.3 Construct

Our primary evaluation metric was the length of the history slice, assuming that a user would be willing to pay higher execution time. However, for completeness, we do compare various techniques in terms of the execution time (or the number of test runs), so that the user can choose the most appropriate configuration for their context.

8 Related Work

Our work intersects with different areas of research. In this section, we compare our dynamic delta refinement algorithm with related work.

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 [4,36,37,47], localize bugs [52, 53], and support predictions [20, 56].

In our earlier work [30,33], we defined the problem of *semantic history slicing* and proposed an algorithm CSlicer which conservatively computes a sub-history that preserves the desired test properties. The advantage of CSlicer is its efficiency – it only executes the tests once and assumes all code entities touched by the tests can potentially affect the test results. Our algorithm has stronger guarantees than CSlicer on slice quality and always returns 1-minimal solutions within a reasonable amount of time. In fact, CSlicer and Definer can be combined so that the output from CSlicer is used as an input to Definer to achieve both precision and efficiency [31].

Another interesting take on history analysis is *history transformation* [19, 36]. Muşlu et al. [36] introduced a concept of *multi-grained* development history views. Instead of using a fixed representation of the change history, the authors 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 semantic history slicing to build a history view which clusters semantically related changes as high-level logical groups. This semantics summarization view [36] is much more meaningful than the original commit-based representations for history understanding and analysis.

8.2 Change Impact Analysis

Change Impact Analysis [1] aims to determine the effects of source code modifications. This 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* [1, 27], *dynamic* [29] and *combined* [43, 54] approaches. The latter category is most closely related to our work. Ren et al. [43] 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 [54] improved Chianti by extending the standard dynamic call graph with field access information.

The invariant deltas we used for locating precise impacts of changes can be viewed as a dynamic impact analysis technique. In fact, we are not limited to using Daikon for this purpose. The performance of Definer may be further improved with a custom lighter-weight runtime tracing technique. Moreover, the local backward analysis which matches affected program points to other related changes belongs to the static impact analysis category. A whole range of static analyses with different levels of precision can be integrated into our algorithm to improve ranking accuracy and performance.

8.3 Dynamic Behavioral Analysis

Through program instrumentation and execution tracing, dynamic analysis techniques [18, 40, 42] allow the comparison of precise runtime program behaviors. Daikon [10] is one example of such techniques which discover likely program invariants from runtime executions. Daikon instruments the target program, traces variables of interest, and infers likely invariants for them. It has been widely used for many software developing tasks, including debugging [3, 8], regression testing [41, 50], bug prevention [9] and more.

DIDUCE [18] is another tool for *dynamic invariant detection*. It trains a model for the target program by formulating hypotheses of invariants obeyed by the program and refining hypotheses dynamically through “presumably-good” runs. The produced model can be used to check for potential errors in other test runs. We use a similar idea of forming and updating hypotheses dynamically with multiple test executions. The key difference is that our goal is to infer change significance rather than program invariants. Therefore, we can exploit useful information from both passing and failing runs to improve the accuracy of our significance model.

Our work is also related to *behavioral regression testing* [24, 40] with respect to the usage of test executions for exposing behavioral differences across program versions. In our analysis, we perform behavioral comparisons with a different goal: behavioral regression testing reports the results of behavioral comparison to users in order to help them complete and improve the quality of existing regression test suites [24, 40], whereas our goal is to speed up history slicing by identifying significant changes with the guidance from the behavioral differences.

8.4 Fault Localization

Delta debugging [52] uses divide-and-conquer-style iterative test executions to narrow down the causes of software failures. Delta debugging has been applied to minimize the set of changes which cause regression test failures. This problem can be considered as finding minimal semantic history slices with respect to the failure-inducing properties. In contrast to Definer which extracts semantic information from test results to guide its subsequent partition, delta debugging does not exploit this information, and its partition scheme is fixed. Regarding slice quality, Zeller and Hildebrandt [53] consider an approximated version of minimality, i.e., *1-minimalily*, which guarantees that removing any single change set breaks the target properties. This trade-off on solution quality enables the authors to use an efficient divide-and-conquer search method.

Selective bisection debugging [46] is an enhancement over the traditional *bisection debugging* approaches based on binary search over software version history (e.g., Git-bisect [14]). Bisection debugging is widely used in practice to identify bug introducing commits, but it can be expensive due to costly compilation and test execution process. Selective bisection debugging uses *test selection* and *commit selection* to reduce the number of tests to run and the number of commits to consider. In particular, commit selection uses test coverage information to predict whether a certain commit in a bisection step does not lead to failing any tests and thus can be skipped to save time. This is similar to Definer as they both rely on test signals to make predictions in the hope to reduce overall computational costs. Conceptually, though, there is a difference: selective bisection debugging tries to find which commits cause the test to fail, while Definer looks for commits that cause the test to pass. Another difference is that bisection debugging tries to locate a certain version while Definer attempts to find a 1-minimal history slice, which is a more difficult problem ($\mathcal{O}(\log n)$ vs. $\mathcal{O}(n^2)$).

9 Conclusion and Future Work

We proposed the dynamic delta refinement algorithm for finding minimal semantic history slices. It relies on change-significance learning techniques that are shown to be effective in speeding up the slicing process when applied to real-world software projects. We also introduced the commit splitting operator which further improves the precision of history slicing by splitting commits from the original change histories into several finer-grained commits. We have implemented the algorithms as a prototype tool, DEFINER, which operates on Java projects hosted in Git. DEFINER greatly improves the precision of the history slices over state-of-the-art techniques, although at the cost of increased execution time.

For future work, we would like to explore the possibility of applying delta refinement to debugging and fault localization. We also see room for improvement in terms of performance by combining different slicing approaches and parallelizing test executions as much as possible.

Acknowledgements This research is partly supported by the Singapore Ministry of Education Academic Research Fund Tier 1 (award No. 2018-T1-002-069).

References

1. Arnold, R.S.: Software Change Impact Analysis. IEEE Computer Society Press, Los Alamitos, CA, USA (1996)
2. Apache Commons Byte Code Engineering Library. `https://commons.apache.org/proper/commons-bcel` (2015)
3. Brun, Y., Ernst, M.D.: Finding Latent Code Errors via Machine Learning over Program Executions. In: Proceedings of the 26th International Conference on Software Engineering, pp. 480–490. IEEE Computer Society, Washington, DC, USA (2004)
4. Brun, Y., Holmes, R., Ernst, M.D., Notkin, D.: Early Detection of Collaboration Conflicts and Risks. IEEE Transactions on Software Engineering **39**(10), 1358–1375 (2013)
5. Chawathe, S.S., Rajaraman, A., Garcia-Molina, H., Widom, J.: Change Detection in Hierarchically Structured Information. In: Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, pp. 493–504 (1996)
6. Apache Commons Compress Library. `https://commons.apache.org/proper/commons-compress` (2018)
7. Apache Commons CSV Library. `https://commons.apache.org/proper/commons-csv` (2017)
8. Dodoo, N., Lin, L., Ernst, M.D.: Selecting, Refining, and Evaluating Predicates for Program Analysis. Tech. Rep. MIT-LCS-TR-914, MIT Laboratory for Computer Science, Cambridge, MA (2003)
9. Ernst, M.D., Cockrell, J., Griswold, W.G., Notkin, D.: Dynamically Discovering Likely Program Invariants to Support Program Evolution. In: Proceedings of the 21st International Conference on Software Engineering, pp. 213–224. ACM, New York, NY, USA (1999). DOI 10.1145/302405.302467
10. Ernst, M.D., Perkins, J.H., Guo, P.J., McCamant, S., Pacheco, C., Tschantz, M.S., Xiao, C.: The Daikon System for Dynamic Detection of Likely Invariants. Science of Computer Programming **69**(1-3), 35–45 (2007)
11. Ferzund, J., Ahsan, S.N., Wotawa, F.: Empirical Evaluation of Hunk Metrics as Bug Predictors. In: Proceedings of the International Conferences on Software Process and Product Measurement, pp. 242–254. Springer-Verlag, Berlin, Heidelberg (2009). DOI 10.1007/978-3-642-05415-0_18
12. Fluri, B., Gall, H.C.: Classifying Change Types for Qualifying Change Couplings. In: Proceedings of the 14th IEEE International Conference on Program Comprehension, pp. 35–45. IEEE (2006)

13. Fluri, B., Wuersch, M., Pinzger, M., Gall, H.: Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction. IEEE Transactions on Software Engineering **33**(11), 725–743 (2007)
14. Git: git-bisect Documentation. `http://git-scm.com/docs/git-bisect` (2016)
15. Git Version Control System. `https://git-scm.com` (2016)
16. Git - Git Basics. `https://git-scm.com/book/en/v2/Getting-Started-Git-Basics` (2018)
17. Git Tools - Interactive Staging. `https://git-scm.com/book/en/v2/Git-Tools-Interactive-Staging` (2018)
18. Hangal, S., Lam, M.S.: Tracking Down Software Bugs Using Automatic Anomaly Detection. In: Proceedings of the 24th International Conference on Software Engineering, pp. 291–301. ACM, New York, NY, USA (2002). DOI 10.1145/581339.581377
19. Hayashi, S., Omori, T., Zenmyo, T., Maruyama, K., Saeki, M.: Refactoring Edit History of Source Code. In: Proceedings of the 28th IEEE International Conference on Software Maintenance, pp. 617–620. IEEE (2012)
20. Herzig, K., Zeller, A.: The Impact of Tangled Code Changes. In: Proceedings of the 10th Working Conference on Mining Software Repositories, pp. 121–130. IEEE Press, Piscataway, NJ, USA (2013)
21. Igarashi, A., Pierce, B.C., Wadler, P.: Featherweight Java: A Minimal Core Calculus for Java and GJ. ACM Transactions on Programming Languages and Systems **23**(3), 396–450 (2001)
22. Apache Commons IO Library. `https://commons.apache.org/proper/commons-io` (2017)
23. JGit: A Lightweight, Pure Java Library Implementing the Git Version Control System. `https://eclipse.org/jgit` (2016)
24. Jin, W., Orso, A., Xie, T.: Automated Behavioral Regression Testing. In: Proceedings of the 2010 3rd International Conference on Software Testing, Verification and Validation, pp. 137–146. IEEE Computer Society, Washington, DC, USA (2010). DOI 10.1109/ICST.2010.64
25. JIRA Software. `https://www.atlassian.com/software/jira` (2017)
26. Kästner, C., Apel, S.: Type-Checking Software Product Lines - a Formal Approach. In: Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering, pp. 258–267. IEEE Computer Society, Washington, DC, USA (2008)
27. Kung, D.C., Gao, J., Hsia, P., Wen, F., Toyoshima, Y., Chen, C.: Change Impact Identification in Object Oriented Software Maintenance. In: Proceedings of the International Conference on Software Maintenance, pp. 202–211. IEEE Computer Society, Washington, DC, USA (1994)
28. Apache Commons Lang Library. `https://commons.apache.org/proper/commons-lang` (2018)
29. Law, J., Rothermel, G.: Whole Program Path-Based Dynamic Impact Analysis. In: Proceedings of the 25th International Conference on Software Engineering, pp. 308–318. IEEE (2003)
30. Li, Y., Rubin, J., Chechik, M.: Semantic Slicing of Software Version Histories. In: Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering, pp. 686–696. Lincoln, NE, USA (2015)
31. Li, Y., Zhu, C., Gligoric, M., Rubin, J., Chechik, M.: Towards Generalized Semantic History Slicing. Tech. rep., Nanyang Technological University (2019)
32. Li, Y., Zhu, C., Rubin, J., Chechik, M.: Precise Semantic History Slicing through Dynamic Delta Refinement. In: Proceedings of the 31 IEEE/ACM International Conference on Automated Software Engineering, pp. 495–506 (2016)
33. Li, Y., Zhu, C., Rubin, J., Chechik, M.: Semantic Slicing of Software Version Histories. IEEE Transactions on Software Engineering **44**(2), 182–201 (2017)
34. Li, Y., Zhu, C., Rubin, J., Chechik, M.: CSlicerCloud: A Web-Based Semantic History Slicing Framework. In: Proceedings of the 40th International Conference on Software Engineering (2018)
35. Mercurial Source Control Management System. `http://mercurial.selenic.com` (2016)
36. Muşlu, K., Swart, L., Brun, Y., Ernst, M.D.: Development History Granularity Transformations. In: Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering, pp. 697–702. Lincoln, NE, USA (2015)

37. Murphy-Hill, E., Parnin, C., Black, A.P.: How We Refactor, and How We Know It. IEEE Transactions on Software Engineering **38**(1), 5–18 (2012)
38. Apache Maven Project. `https://maven.apache.org` (2015)
39. Apache Commons Net Library. `https://commons.apache.org/proper/commons-net` (2017)
40. Orso, A., Xie, T.: BERT: BEhavioral Regression Testing. In: Proceedings of the 2008 International Workshop on Dynamic Analysis: Held in Conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 36–42. New York, NY, USA (2008)
41. Pastore, F., Mariani, L., Hyvärinen, A.E.J., Fedyukovich, G., Sharygina, N., Sehestedt, S., Muhammad, A.: Verification-Aided Regression Testing. In: Proceedings of the 2014 International Symposium on Software Testing and Analysis, pp. 37–48. ACM, New York, NY, USA (2014). DOI 10.1145/2610384.2610387
42. Perkins, J.H., Ernst, M.D.: Efficient Incremental Algorithms for Dynamic Detection of Likely Invariants. In: Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering, pp. 23–32. ACM, New York, NY, USA (2004). DOI 10.1145/1029894.1029901
43. Ren, X., Shah, F., Tip, F., Ryder, B.G., Chesley, O.: Chianti: A Tool for Change Impact Analysis of Java Programs. In: Proceedings of the 19th aAnual ACM SIGPLAN conference on Object-oriented Programming, Systems, Languages, and Applications, pp. 432–448. ACM, New York, NY, USA (2004)
44. Rothermel, G., Harrold, M.J.: Analyzing Regression Test Selection Techniques. IEEE Transactions on Software Engineering **22**(8), 529–551 (1996). DOI 10.1109/32.536955
45. Rubin, J., Kirshin, A., Botterweck, G., Chechik, M.: Managing Forked Product Variants. In: Proceedings of the 16th International Software Product Line Conference, vol. 1, pp. 156–160. ACM, New York, NY, USA (2012)
46. Saha, R., Gligoric, M.: Selective Bisection Debugging. In: Proceedings of the 20th International Conference on Fundamental Approaches to Software Engineering, pp. 60–77. Springer-Verlag New York, Inc., New York, NY, USA (2017). DOI 10.1007/978-3-662-54494-5_4
47. Servant, F., Jones, J.A.: History slicing. In: Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering, pp. 452–455 (2011)
48. Apache Subversion (SVN) Version Control System. `http://subversion.apache.org` (2016)
49. Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., Sundaresan, V.: Soot - a Java Bytecode Optimization Framework. In: Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research, pp. 125–135. IBM Press (1999)
50. Xie, T., Notkin, D.: Checking inside the Black Box: Regression Testing by Comparing Value Spectra. IEEE Transactions on Software Engineering **31**(10), 869–883 (2005)
51. YAML Ain't Markup Language. `http://www.yaml.org/` (2017)
52. Zeller, A.: 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, pp. 253–267. Springer-Verlag, London, UK, UK (1999)
53. Zeller, A., Hildebrandt, R.: Simplifying and Isolating Failure-Inducing Input. IEEE Transactions on Software Engineering **28**(2), 183–200 (2002)
54. Zhang, L., Kim, M., Khurshid, S.: Localizing Failure-Inducing Program Edits Based on Spectrum Information. In: Proceedings of the 27th International Conference on Software Maintenance, pp. 23–32. IEEE (2011)
55. Zhu, C., Li, Y., Rubin, J., Chechik, M.: A Dataset for Dynamic Discovery of Semantic Changes in Version Controlled Software Histories. In: Proceedings of the 14th International Conference on Mining Software Repositories, pp. 523–526. IEEE Press, Piscataway, NJ, USA (2017). DOI 10.1109/MSR.2017.49
56. Zimmermann, T., Weißgerber, P., Diehl, S., Zeller, A.: Mining Version Histories to Guide Software Changes. In: Proceedings of the 26th International Conference on Software Engineering, pp. 563–572. IEEE Computer Society, Washington, DC, USA (2004)