Precise Semantic History Slicing through Dynamic Delta Refinement

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

Chenguang Zhu
University of Toronto
Toronto, ON, Canada
czhu@cs.toronto.edu

Julia Rubin
MIT
Cambridge, MA, USA
mjulia@csail.mit.edu

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

ABSTRACT

Semantic history slicing solves the problem of extracting changes related to a particular high-level functionality from the 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 *minimal* 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. 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.

CCS Concepts

•**Software and its engineering** → **Software configuration management and version control systems;** *Dynamic analysis;* Software evolution;

Keywords

Semantic history slicing, program analysis, dynamic invariants, software configuration management.

1. INTRODUCTION

Software Configuration Management systems (SCMs) are widely used in software development practices. These systems, e.g., Git [8], SVN [5], Mercurial [10], are useful for capturing incremental changes made by developers, examining or reverting changes, identifying developers responsible for a specific change, creating development streams, and more. Incremental changes are manually grouped by developers to form *commits* (a.k.a. *change sets*). Commits are stored sequentially and ordered by their time stamps, so that it is convenient to trace back to any version in the history.

Yet, the sequential organization of changes is inflexible and lacks support for many tasks that require high-level, semantic understanding of program functionality [36, 29]. For example, developers often need to locate and transfer functionality from one branch to another: either for porting bug fixes or for splitting large chunk commits into multiple functionally-independent pull requests. Identifying failure-inducing changes in version histories is another challenge that developers face in their work.

Semantic History Slicing. *Semantic history slicing* identifies a set of commits in a change history that relate to each other based on a certain criterion. For example, CSlicer [27] identifies and extracts a set of functionally-related commits that correspond to a specific high-level functionality. The functionality is defined by a test suite, and the sliced history has to be perfectly functional and pass all the tests in the suite. *Git-bisect* [7] and *delta debugging* [42] 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 programs in the search space, i.e., those that are induced by all subsets of commits in the history. A valid semantic history slice has to be efficiently discovered from the exponential number of candidates.

Existing solutions approach this problem from two different angles. CSlicer [27] 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 results in potential inclusion of unnecessary or irrelevant changes into the history slice.

ASE'16, September 3–7, 2016, Singapore, Singapore
© 2016 ACM. 978-1-4503-3845-5/16/09...$15.00
http://dx.doi.org/10.1145/2970276.2970336

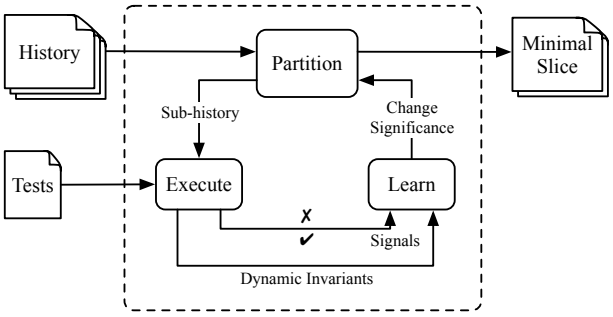


Figure 1: Dynamic delta refinement loop.

Divide-and-conquer-style techniques, such as delta debugging, guarantee accuracy of the result. Yet, they can be very expensive to run as they execute the test suite multiple times, depending on the way history is partitioned and on the order in which partitions are tested.

Dynamic Delta Refinement. In this paper, we propose a precise semantic history slicing technique based on *iterative refinement* and *change significance ranking*. We discover relevance of changes to the target tests through successive test runs and utilize this information to guide the history partition and speed up the refinement process. We refer to this technique as *dynamic delta refinement*. Its key insight is that by comparing the runtime executions of two program versions – before and after a change, we can extract information about the precise impact of the changes at various program points. By combining impact information with test outcomes (pass or fail), we are able to accurately infer the significance of changes with respect to the target tests. In particular, if the tests still pass after removal of a change, then the removed change and its family of related changes are insignificant to the tested functionalities. We give more details on how such families of changes can be detected using dynamic program invariants generated by Daikon [16] in Sec. 4.

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

Contributions.

1. We show how dynamic delta refinement can learn significance of changes with respect to a specific high-level functionality.
2. We define minimal semantic slices and describe an algorithm for computing them based on dynamic delta refinement.
3. We report on an implementation of a fully-automated precise semantic history slicing tool that operates on Java projects hosted in Git repositories.
4. We compare our technique with previous work in terms of precision and efficiency. Based on empirical evaluation, our technique achieves ∼46% improvement in accuracy compared with CSlicer. We also demonstrate the advantage of using change significance to speed up the basic partition scheme used by delta debugging.

Organization. The rest of this paper is organized as follows. Sec. 2 illustrates how dynamic invariants are used for learning change significance and how significance ranking is used to guide history partition. Sec. 3 provides the necessary background for the rest of the paper. In Sec. 4, we formalize the delta refinement algorithm for finding minimal semantic slices and prove its correctness. In Sec. 5, we describe our implementation and experimental results. Finally, we discuss related work and conclude in Sec. 6 and 7, respectively.

2. OVERVIEW OF THE APPROACH

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

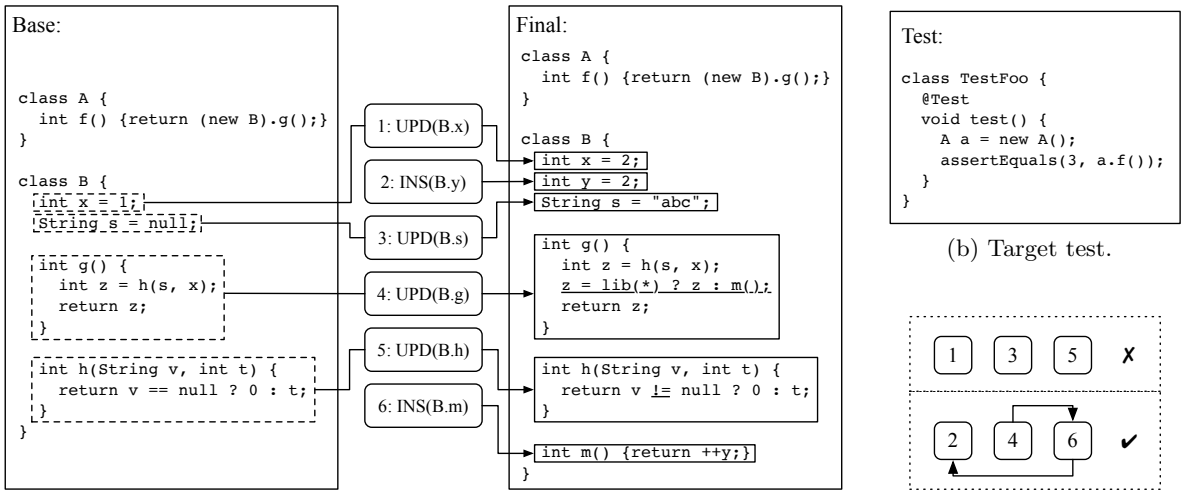
2.1 Changes and Dependencies

Example 1. Fig. 2a shows two versions of a Java program `Foo.java`: "base" and "final". The "final" version introduces a few modifications to the class `B` through a series of *atomic changes*. Atomic changes are defined over the *abstract syntax trees* (ASTs) of the program as *insertions* (Ins), *deletions* (Del), or *updates* (Upd) of AST nodes (e.g., fields, methods, etc.) Specifically, there are six atomic changes between the "base" and the "final" versions (listed in no particular order), ①: an update to the field `B.x`; ②: an insertion of a new field `y` into the class `B`; ③: an update to the field `B.s`; 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`; ⑤: an update to method `B.h()`, which replaces "`==`" by "`!=`"; and ⑥: an insertion of a new method `m()` into the class `B`.

The `lib(*)` method called in `g()` represents an external library whose returned value is only known during runtime. We do know that the library method behaves deterministically but cannot predict its return value without executing it. The desired functionality of the program is captured by a unit test for `Foo.java` which asserts that the returned value of the method `A.f()` should be equal to 3 (see Fig. 2b). We denote this test by T. Note that the test assertion holds in the final version of the program but fails in the base one.

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

Change Dependencies. Atomic changes are not completely independent from each other. In order to construct a well-formed program, some changes have to be applied



(a) Atomic changes. (b) Target test. (c) Change dependencies.

Figure 2: Atomic changes between the "base" and "final" versions of `Foo.java`.

as prerequisites for others [35, 27]. For example, INS(`B.m`) depends on INS(`B.y`) since the method `B.m()` accesses the field `B.y` and requires the declaration of the field in order to compile; and UPD(`B.g`) depends on INS(`B.m`) since the new version of the method `B.g()` invokes `B.m()` (see Fig. 2c).

We are only interested in producing well-formed programs. The partition of changes thus has to obey the dependency relations. That is, reverting a subset of atomic changes results in a well-formed program only if the remaining changes have all their dependencies satisfied. The change dependencies can be computed systematically, as we describe in Sec. 5.1.

2.2 Learning Change Significance

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

In Example 1, 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* [16]: likely invariants that may not generalize but that hold for the exercised test executions. For simplicity, we refer to them as invariants from now on. For instance, "`B::x == 2`" is a field invariant which indicates that the value of the field `B.x` equals to 2 during the initial execution. Another example is "`A.f()::return == 3`" which is a method post-condition asserting that the return value of `A.f()` is 3.

We denote by H^- the set of reverted changes, I the initial set of invariants for the final version of the program, and I' the invariants after changes are reverted. The row H^- of Fig. 3 shows four possible cases of reverted changes in Example 1. The deltas in the generated invariants before and after reverting changes is shown in row "$I \setminus I'$" of the table. The rows "$T(H^+)$" and "Signals" show the test outcomes and significance signals learned for each case, respectively. We discuss each case in turn below.

Case 1: Test Passing without Extra Signal. Suppose a singleton atomic change set $H^- = \{\boxed{1}\}$ is reverted during the first partition step, and the new program is now equivalent to applying $H^+ = \{\boxed{2}, \boxed{3}, \boxed{4}, \boxed{5}, \boxed{6}\}$ to the base version. The declarations and initializations of `x` are reverted to the base version: `x` initialized to 1 instead of 2.

Static analysis is unable to determine whether this change would affect the test results due to the undetermined returns of `lib(*)`. But we are able to precisely detect the *impact* of reverting $\boxed{1}$ by comparing the new set of invariants I' generated during the actual execution of the new program to the original invariants I. In this case, we observe that only one invariant disappears after reverting $\boxed{1}$, namely, "`B::x == 2`" (see row "$I \setminus I'$" in Fig. 3). This indicates that the impact of reverting $\boxed{1}$ is local to the change itself and does not flow into other program points.

In fact, `lib(*)` returns `false` at runtime and thus the change on `B.x` does not propagate through – the returned value from `h(s,x)` is overwritten by `m()` which is independent of the change. The test outcome is unchanged and therefore the value of `B.x` is considered insignificant. We decrease the significance score of $\boxed{1}$ (denoted by $\downarrow$ in Fig. 3 row "Signals").

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

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

Case 3: Test Failing by Determined Causes. When $\boxed{4}$ is reverted, the conditional assignment statement "`z = lib(*) ? z : m()`" in `g()` is removed. The test fails because now the value from `h(s,x)` flows through, which is different from the old value from `m()`. Since an atomic change is already the smallest unit in our analysis, we can pinpoint $\boxed{4}$ as the definite cause of the test failure.

All invariants violated by the revert are directly impacted by the change and most likely cause the failure. They are

	Case 1	Case 2	Case 3	Case 4
H^-	[1]	[1] [3]	[4]	[3] [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	[1]↓	[1]↓ [3]↓ [5]↓	[2]↑ [4]↑ [6]↑	

Figure 3: Change significance learning case by case.

as follows: "`B::y one of 2, 3`" which asserts that the field `y` used to take both 2 and 3 (now `y` can only be 2), and "`B.m()::return == 3`" which asserts that the return of `m()` used to be 3 (now `m()` does not return at all). Therefore, we consider both [2] and [6], which are associated with `B.y` and `B.m()`, respectively, significant for the test.

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

2.3 History Partition by Significance Ranking

The basic idea of history partition is inspired by delta debugging [42]. In the first iteration, the history is split into two halves which are then tested individually. If one of the partitions passes the test, then the process continues recursively on the successful partition. Otherwise, less aggressive partitions are produced by reverting fewer changes and keeping more. For example, we can split the history into four similar-size change sets and revert each of them, one at a time. If none of the attempts are successful, then finer-grained partitions are produced until we reach a point where only a single atomic change is reverted at a time. Then we are able to classify the change precisely according to the test results. The process terminates when a *1-minimal* [42] history slice is found: reverting any single change in the slice fails the test.

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 predict compilation errors without needing to compile the program.

Example 2. We use another example with a slightly more complex change history to illustrate our enhanced history partition scheme. In this example, there are eight atomic changes $\{[1], \ldots, [8]\}$, adding two non-essential changes [7] and [8] on top of the history in Example 1. The set of essential changes is still $\{[2], [4], [6]\}$. Due to space limitations, we omit the details of the additional changes[1].

The actual steps taken when analyzing this example are shown in Fig. 4. During the first step ($n = 1$), the history

[1]The code and detailed changes for Example 2 can be found at: bitbucket.org/liyistc/gitslice/wiki/ase16-e2.

n	Partition (H^+, H^-)	T	Signals
1	[1] [2] [3] [4] [5] [6] [7] [8]	-	
	[1] [2] [3] [4] 5 6 7 8	-	
2	[1] [2] [3] [4] [5] [6] [7] [8]	-	
	[1] [2] [3] [4] [5] [6] [7] [8]	✘	
	[1] [2] [3] [4] 5 6 7 8	-	
	[1] [2] [3] [4] [5] [6] 7 8	✔	[3]↓[5]↓[7]↓[8]↓
3	[1] [2] [3] [4] 5 6	✔	[3]↓[5]↓
4	[1] [2] [4] [6]	-	
	[1] [2] 4 6	✘	
5	[1] [2] [4] [6]	✔	[1]↓
6	2 [4] [6]	-	
	[2] 4 [6]	✘	[2]↑[4]↑[6]↑
	[2] [4] 6	-	

Figure 4: Enhanced history partition scheme.

is partitioned into two equal halves, i.e., $\{[1], [2], [3], [4]\}$ and $\{[5], [6], [7], [8]\}$. We keep one set and revert the other but only to find that the dependencies [6] → [2] and [4] → [6] are violated. In Fig. 4, change dependency violations are represented by "-" in column "T". No test run is needed so far.

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

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

For this example, six test runs are needed for finding the minimal solution using the enhanced partition scheme. In contrast, the basic partition scheme from [42], without significance learning or change dependency analysis, requires thirteen test runs and twelve additional (failed) compilations.

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

Figure 5: Language syntax rules [22].

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

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

Figure 6: Types of atomic changes [17].

3. PRELIMINARIES

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

Language Syntax. To keep the presentation concise, we step back from the complexities of the full Java language and concentrate on the core object-oriented features. We adopt a simple functional subset of Java from *Featherweight Java* [22], denoting it by P. The syntax rules of the language P are given in Fig. 5. Many advanced Java features, e.g., interfaces, abstract classes and reflection, are removed from P, while the typing rules which are crucial for the compilation correctness are retained [24].

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

Abstract Syntax Tree. A valid program $p \in P$ can be parsed as an *abstract syntax tree* (AST), denoted by $\textsc{Ast}(p)$. We adopt a simplified AST model where the smallest entity nodes are fields and methods. Formally, $r = \textsc{Ast}(p)$ is a rooted tree with a set of nodes $V(r)$. The root of r is denoted by $\textsc{Root}(r)$ which represents the compilation unit, i.e., the program p. Each entity node x has an identifier and a value, denoted by $id(x)$ and $\nu(x)$, respectively. In a valid AST, the identifier for each node is unique (e.g., fully qualified names in Java), and the values are canonical textual representations of the corresponding entities. We denote the parent of a node x by $\textsc{Parent}(x)$. Fig. 7 shows an AST for the base version of the program `Foo.java` from Fig. 2a.

Change and Change History. Let Γ be the set of all ASTs. An *atomic change* operation $\delta : \Gamma \to \Gamma$ is either an *insert*, *delete* or *update* (see Fig. 6). It transforms $r \in \Gamma$ producing a new AST r' such that $r' = \delta(r)$. An insertion $\textsc{Ins}((x,n,v),y)$ inserts a node x with an identifier n and a value v as a child of a node y. A deletion $\textsc{Del}(x)$ removes the node x from the AST. An update $\textsc{Upd}(x,v)$ replaces the node x with the node v. A change operation is *applicable* on an AST if its preconditions are met. For example, the insertion $\textsc{Ins}((x,n,v),y)$ is applicable on r if and only if $y \in V(r)$. Insertion of an existing node is treated the same as an update. Δ denotes the set of all atomic changes.

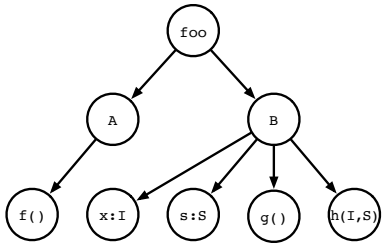


Figure 7: AST of `Foo.java` at the base version.

A *history* is a sequence of changes $H = \langle\delta_1, \ldots, \delta_k\rangle$. A *sub-history* is a sub-sequence of a history, i.e., a sequence derived by removing changes from H without altering the ordering. We write $H' \subseteq H$ indicating that H' is a sub-history of H and refer to $\langle\delta_i, \ldots, \delta_j\rangle$ as $H_{i..j}$. A change history $H = H_{-1} \circ \delta_1$ is *applicable to* r if δ_1 is applicable to r and H_{-1} is applicable to $\delta_1(r)$.

Test Cases. We assume that semantic functionalities can be captured by test cases and the execution trace of a test case is deterministic. A *test case* t is a function $t : P \to \mathbb{B}$ such that for a given program $p \in P$, $t(p)$ is true if and only if the test succeeds, and false otherwise. A *test suite* is a collection of unit tests that can exercise and demonstrate the functionality of interest. Let test suite T be a set of test cases $\{t_i\}$. We write $p \models T$ if and only if program p passes all tests in T, i.e., $\forall t \in T \cdot t(p)$.

Dynamic Invariants. *Dynamic invariants* [34] 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: (1) method preconditions asserting values of input parameters, (2) method postconditions asserting returned values, and (3) all values taken by fields throughout the execution.

A wide range of dynamic invariants is detected and reported by Daikon [16], but not all of them are useful for inferring change significance. In particular, a failing invariant which depends on two or more variables has ambiguous causes: altering either one of them can break the invariant. Therefore, we only 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, $I \setminus I' = \{i \in I | \neg(\exists i' \in I' \cdot i' \Rightarrow i)\}$.

Precise Semantics-preserving Slice. Consider a program $p_0 \in P$ and its k subsequent versions $p_1, \ldots, p_k$ such that they are all well-formed. 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 $0 \le i \le k$. Let T be a set of tests passed by p_k, i.e., $p_k \models T$.

Definition 1. *(Semantics-preserving slice [27]). 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 prop-

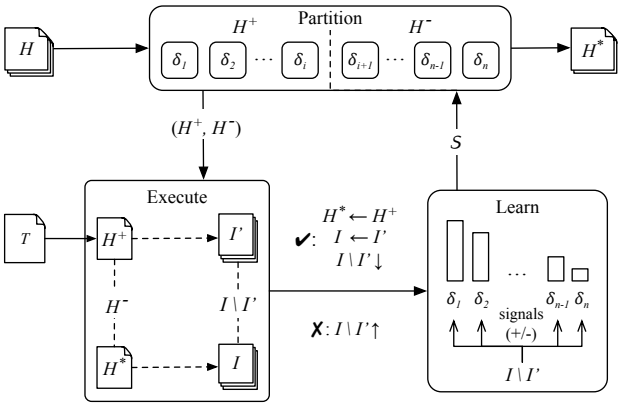


Figure 8: Dynamic delta refinement overview.

erties. However, the optimality of the sliced history cannot always be guaranteed by polynomial-time algorithms [27]: 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 2. *(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. ALGORITHM

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

4.1 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 Fig. 8. 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 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) $\mathcal{S} : \Delta \to \mathbb{R}$, the change significance ranking – a function from atomic changes to real numbers, updated according to the outcomes from the execution phase. The algorithm is presented in Fig. 9 as a set of generic rules specifying the minimal requirements for each phase.

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

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.

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:

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

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

Execution and Learning:

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

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

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

Figure 9: The dynamic delta refinement algorithm.

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

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

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

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

When H^+ passes T, the Pass rule applies. 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 [11]. For each affected method postcondition, we collect all statements within the method body that have potential impacts on the method return (e.g., changed value flows into the return). For instance, using a simple backward data-flow analysis, the invariant "`B.g()::return == 3`" in Example 1 is matched to (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 $|H^-|$ is greater than one, as discussed before, the cause of test failure is not determined. We do not infer change significance in this case (FAIL-2). Otherwise, we perform a similar analysis as in PASS and increase the significance scores of the related changes (FAIL-1).

Termination Condition. The algorithm never attempts the same partition (H^+) twice and it terminates whenever H^* becomes empty or the 1-minimal condition defined in Definition 2 is met – $\forall \delta \in H^* \cdot (H^* \setminus \{\delta\}) \not\models T$.

4.2 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.*

To see this, suppose the algorithm does no terminate. Since H^* has finite number of changes initially and its length is monotonically decreasing, $|H^*|$ has to eventually stay constant. Because of the fairness condition, every atomic change in H^* is eventually reverted and tested. If none of the tests pass, then the 1-minimal condition is met. If one of the tests passes, then $|H^*|$ should decrease. Both cases lead to contradictions.

5. EVALUATION

In this section, we describe our implementation of a minimal semantic slicing tool based on dynamic delta refinement algorithm. We evaluate our implementation w.r.t. both its precision and performance using a benchmark suite obtained from real open source software repositories.

5.1 Implementation

We implemented the delta refinement algorithm as a fully-automated precise semantic history slicing tool, DEFINER, for Java projects hosted in Git repositories. We describe the implementation and some of the applied optimizations below.

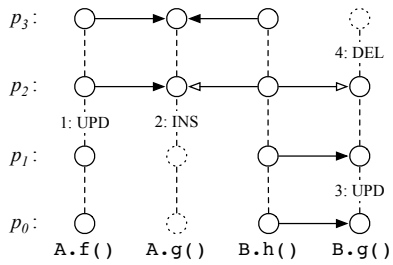


Figure 10: Analyzing change dependencies.

Change Dependency Analysis. To avoid running into compilation errors, we perform a pre-analysis for each version in the history and compute direct dependencies for all changed AST nodes. This analysis produces a *multi-version change dependency graph* as shown in Fig. 10.

In this example, there are four program versions, i.e., p_0, p_1, p_2 and p_3, all of which are well-formed. There are three changed nodes, i.e., methods `A.f()` and `A.g()` which belong to class `A`, as well as `B.g()` which belongs to class `B`. There is also a fixed node `B.h()` which stays unchanged. Class `B` is a sub-class of `A`. Each node has a separate time-line on which its changes are labeled. In particular, `A.f()` has an update between p_1 and p_2; `A.g()` is inserted between p_1 and p_2; and `B.g()` is updated after p_0 but deleted after p_2. In Fig. 10, solid arrows represent *necessary* dependencies while empty arrows represent *sufficient* dependencies. For instance, a method invocation of `g()` in `B.h()` makes `B.h()` necessarily depend on `B.g()` before p_2. But when `g()` is introduced in the super-class `A` in version p_2, both definitions of `g()` are sufficient dependencies of `B.h()`, i.e., existence of either one of them would satisfy the compilation requirement due to method inheritance.

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

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

Git Adaptation. The generic algorithm discussed in Sec. 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 dependencies between commits which are also known as the *hunk dependencies* [27]. Hunk dependencies for a commit are prerequisites which have to be in place so that the target commit can be applied without causing a Git merge conflict. Partitions of commits which do not comply with the hunk dependencies are discarded immediately without running any tests.

Lazy Dynamic Tracing. The software projects we experimented with are relatively large in size (30 to 190 KLOC). Instrumenting the entire project and tracing test execution

Table 1: Statistics of tested software projects.

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

end-to-end is often impractical. But since we are only interested in the local direct impacts of changes, which give clearer signals for significance, we can simply trace classes that have changed during the input history.

Tooling. We use JGit [9], a Java implementation of Git, for repository manipulation and commit-level hunk dependency analysis [27]. We use a modified version of ChangeDistiller [18] for extracting AST-level atomic changes from Git commits. We also use the Apache Byte Code Engineering Library (BCEL) [1] to analyze dependencies among atomic changes, Daikon [16] for dynamic invariant detection, and Soot [39] for performing local change impact analysis.

DEFINER is written in Java and yields fully-automated analysis of projects built with Maven [4]. The source code of DEFINER and all benchmarks used in our experiments are available online: bitbucket.org/liyistc/gitslice.

5.2 Experiments

The goal of our empirical evaluation of DEFINER is to answer the following research questions:

RQ1 How does the *precision* of history slices produced by DEFINER compare with those produced by CSLICER?

RQ2 How *effective* is change significance ranking 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?

5.2.1 Subjects

We tested DEFINER on a benchmark consisting of eight target functionalities selected from three open source projects, namely Apache Commons IO Library (`io`) [3], Apache Commons Collections Library (`collections`) [2], and Apache Commons Mathematics Library (`math`) [6]. These projects are all written in Java and their development histories are freely accessible online. They are also actively developed and maintained so that there are an abundance of new functionalities (e.g., features, bug fixes and improvements) to choose from. Statistics about each project is shown in Table 1. Columns "#Files" and "LOC" show the number of Java files and the total lines of code, respectively. Column "#C-1y" shows the number of commits between 2015-1-1 and 2016-1-1 for each project. Column "#C-4m" shows the average number of commits over a 4-month period.

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

According to our experience, the lifetime of a functionality typically spans around 100 commits which correspond to a period of 1-4 months for a project under active development.

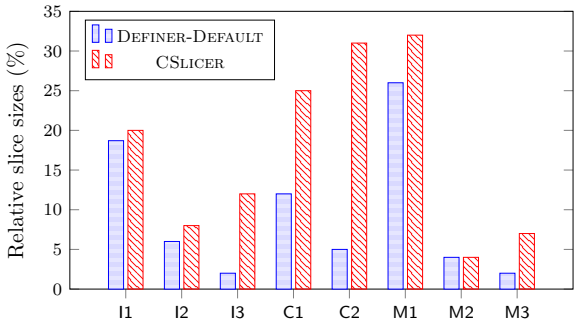


Figure 11: Sizes of slices: Definer vs. CSlicer.

Therefore, the length of H was decided based on two factors: (1) each program version in the chosen history should be well-formed and compilable as assumed by our change dependency analysis, and (2) the average history length of a subject matches the average number of commits during a 4-month development period (∼76.1 commits).

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

The details about each experimental subject are given in Table 2. Column "ID" lists the subject identifiers. Column "End Point" shows the target commits which correspond to the final version in our analysis scope. Columns "$|H|$" and "$|T|$" show the length of the original history segments and the sizes of the target test suites, respectively. Column "$|H^*|$" shows the length of the verified 1-minimal history slice.

5.2.2 Results

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

Experiment 1. 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 (DEFAULT) which adopts a simple partition scheme that reverts negative scored commits first whenever possible. The relative slice size for each subject is computed as the length of the produced history slice divided by the original history length, i.e., $|H^*|/|H|$. The results of the comparison are shown in Fig. 11.

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

Experiment 2. The second experiment evaluates the effectiveness of using change significance ranking and change

Table 2: Experimental subject details and descriptions.

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

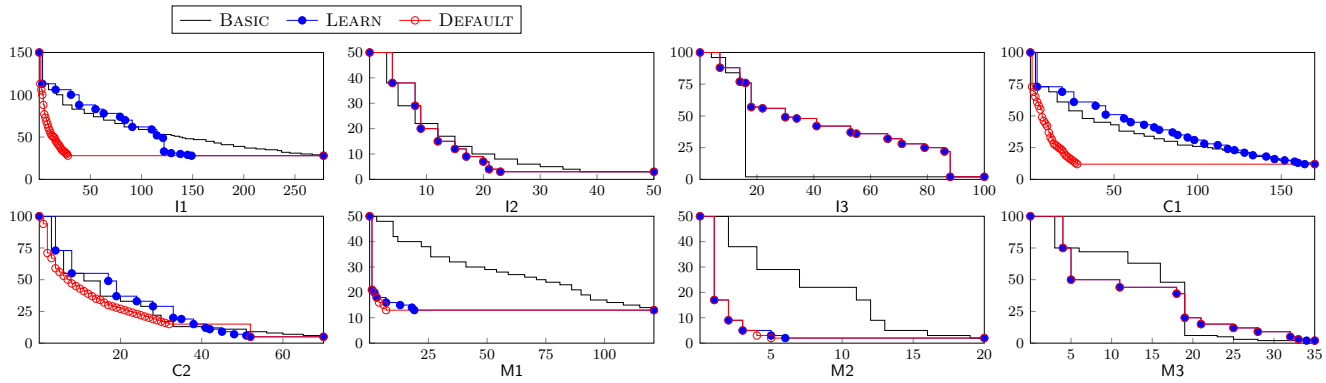


Figure 12: History reduction per test run.

dependency analysis in speeding up the delta refinement loop. We compared three configurations of Definer in terms of the number of test runs needed to achieve 1-minimal solutions: (1) Default as described before, (2) Learn which only enables change significance learning and disables compilation failure prediction based on change dependency analysis, and (3) Basic which also disables significance learning and thus is effectively equivalent to delta debugging which applies the basic (random) partition scheme[2]. All configurations still apply hunk dependency analysis which predicts Git merge failures without actually picking the commits.

The results of the comparisons are shown in Fig. 12 where the length of H^* (y-axis) is plotted as a function of the number of test runs (x-axis) used so far. In general, Default and Learn require fewer test runs than Basic to reach the minimal solution. In most of the cases, the advantage of significance learning is obvious, especially for cases such as I1, M1 and M2 where Learn requires on average only about 33% of test runs compared with Basic. In addition, change dependency analysis which prevents test from running on non-compilable programs helped extend this advantage further – it only takes about 14% of test runs.

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

[2]We could not directly compare with the original implementation in [41] which does not work with Git repositories.

Table 3: Comparisons of different partition schemes.

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

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

All schemes follow the general steps described in Sec. 2.3 with different partition priorities at the beginning of each iteration. The Neg scheme only reverts commits which have negative scores. It is the most conservative one among the three. NonPos is the most aggressive one which reverts all commits with non-positive scores. Low-3 always reverts the lowest one third of the commits according to their significance ranking. Combined attempts all three partitions whenever possible. The results of this experiment can be seen in Table 3, where each column lists the number of test runs required to reach the minimal solution and the best configurations for each row are in bold. All three partition schemes perform well on some of the subjects. For example, Low-3 required the smallest number of test runs for I1, I2 and C2. The combined scheme achieved the best overall performance by winning in 5 out of 8 examples.

5.2.3 *Summary*

To summarize, we evaluated the precision and performance of Definer empirically on a benchmark set of real-world

software projects. We demonstrated that DEFINER produces more precise history slices than existing state-of-the-art techniques, such as CSLICER. Moreover, in the majority of cases, it outperforms the basic partition scheme used by delta debugging, thanks to the change significance ranking learned during the refinement process. With all optimizations combined, DEFINER achieves precise slicing results in an efficient manner.

6. RELATED WORK

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

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 [27, 29, 30, 13, 37], localize bugs [41, 42], and support predictions [45, 21].

Li et. al [27] 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 on slice quality and always returns 1-minimal solutions within a reasonable amount of time. In fact, the two techniques can be combined together so that the output from CSLICER is used as an input to DEFINER to achieve both precision and efficiency.

The goal of *delta debugging* [42, 28] is to simplify and isolate a small test case from a large set of changes which can still manifest the target failures. This problem can be considered as semantic slicing with respect to the failure-inducing properties. Our delta refinement techniques use a similar divide-and-conquer partition process but improve its performance with the significance ranking inferred from previous iterations.

Another interesting take on history analysis is to create flexible views of the change history at various granularities instead of using the fixed revision centric representation. Some notable approaches include *history slicing* [37, 38] and *history transformation* [20, 29]. The promise of these techniques is to provide users the most convenient and effective ways of interacting with change histories and better facilitate the specific software evolution tasks at hand. For example, Muşlu et. al [29] introduced the concept of semantics summarization view which clusters original sequence of commits into semantically related high-level logical groups. Such history transformation operators can be instantiated with our techniques to produce high quality history visualization and understanding tools.

Dynamic Behavioral Analysis. Through program instrumentation and execution tracing, dynamic analysis techniques [19, 34, 32] allow the comparison of precise runtime program behaviors. Daikon [16] 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 [12, 14], regression testing [40, 33], bug prevention [15] and more.

DIDUCE [19] 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* [32, 23] w.r.t. the usage of test executions for exposing behavioral differences across program versions. We differ in how the differences are used: [32, 23] report comparison results to users in order to help them complete and improve the quality of existing regression test suites, whereas our goal is to identify significant changes with the guidance from the behavioral differences.

Change Impact Analysis. *Change Impact Analysis* (IA) [11, 35, 26, 44, 43] solves the problem of determining the effects of source code modifications. It usually means selecting a subset of tests from a regression test suite that might be affected by a given change, or, given a test failure, deciding which changes might be causing it.

Research on IA can be roughly divided into three categories: the *static* [25, 11], *dynamic* [26] and *combined* [31, 35, 43] approaches. The work most related to ours is on the combined approaches to IA. Ren et al. [35] introduced a tool called Chianti for IA of Java programs. Chianti takes two versions of a Java program and a set of tests as the input. First by tracing test executions, it builds dynamic call graphs for both versions before and after the changes. 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 [43] 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 IA technique. In fact, we are not limited to using Daikon for this purpose. The performance of DEFINER can be further improved with a custom lighter-weight runtime tracing technique. Moreover, the backward analysis which matches affected program points to other related changes belongs to the static IA technique category. A whole range of static analyses with different levels of precision can be integrated into our algorithm to trade-off between ranking accuracy and performance.

7. CONCLUSION

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

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.

8. REFERENCES

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

[2] Apache Commons Collections. https://commons.apache.org/proper/commons-collections.

[3] Apache Commons IO. https://commons.apache.org/proper/commons-io.

[4] Apache Maven Project. https://maven.apache.org.

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

[6] Commons Math: The Apache Commons Mathematics Library. https://commons.apache.org/proper/commons-math.

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

[8] Git version control system. https://git-scm.com/.

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

[10] Mercurial source control management system. http://mercurial.selenic.com/.

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

[12] Y. Brun and M. D. Ernst. Finding Latent Code Errors via Machine Learning over Program Executions. In *Proc. of ICSE'04*, pages 480–490, 2004.

[13] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin. Early Detection of Collaboration Conflicts and Risks. *IEEE Trans. Softw. Eng.*, 39(10):1358–1375, Oct. 2013.

[14] N. Dodoo, L. Lin, and M. D. Ernst. Selecting, Refining, and Evaluating Predicates for Program Analysis. Technical Report MIT-LCS-TR-914, MIT Laboratory for Computer Science, Cambridge, MA, July 2003.

[15] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically Discovering Likely Program Invariants to Support Program Evolution. In *Proc. of ICSE'99*, pages 213–224, 1999.

[16] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon System for Dynamic Detection of Likely Invariants. *Sci. Comput. Program.*, 69(1-3):35–45, Dec. 2007.

[17] B. Fluri and H. C. Gall. Classifying Change Types for Qualifying Change Couplings. In *Proc. of ICPC'06*, pages 35–45, 2006.

[18] B. Fluri, M. Wuersch, M. Pinzger, and H. Gall. Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction. *IEEE Trans. Softw. Eng.*, 33(11):725–743, Nov. 2007.

[19] S. Hangal and M. S. Lam. Tracking Down Software Bugs Using Automatic Anomaly Detection. In *Proc. of ICSE'02*, pages 291–301, 2002.

[20] S. Hayashi, T. Omori, T. Zenmyo, K. Maruyama, and M. Saeki. Refactoring Edit History of Source Code. In *Proc. of ICSM'12*, pages 617–620, September 2012.

[21] K. Herzig and A. Zeller. The Impact of Tangled Code Changes. In *Proc. of MSR'13*, pages 121–130, 2013.

[22] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: A Minimal Core Calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, May 2001.

[23] W. Jin, A. Orso, and T. Xie. Automated Behavioral Regression Testing. In *Proc. of ICST'10*, pages 137–146, 2010.

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

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

[26] J. Law and G. Rothermel. Whole Program Path-Based Dynamic Impact Analysis. In *Proc. of ICSE'03*, pages 308–318, 2003.

[27] Y. Li, J. Rubin, and M. Chechik. Semantic Slicing of Software Version Histories. In *Proc. of ASE'15*, pages 686–696, 2015.

[28] G. Misherghi and Z. Su. HDD: Hierarchical Delta Debugging. In *Proc. of ICSE'06*, pages 142–151, 2006.

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

[30] E. Murphy-Hill, C. Parnin, and A. P. Black. How We Refactor, and How We Know It. *IEEE Trans. Softw. Eng.*, 38(1):5–18, Jan 2012.

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

[32] A. Orso and T. Xie. BERT: BEhavioral Regression Testing. In *Proc. of WODA'08*, pages 36–42, 2008.

[33] F. Pastore, L. Mariani, A. E. J. Hyvärinen, G. Fedyukovich, N. Sharygina, S. Sehestedt, and A. Muhammad. Verification-aided Regression Testing. In *Prof. of ISSTA'14*, pages 37–48, 2014.

[34] J. H. Perkins and M. D. Ernst. Efficient Incremental Algorithms for Dynamic Detection of Likely Invariants. In *Proc. of FSE'04*, pages 23–32, 2004.

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

[36] J. Rubin, A. Kirshin, G. Botterweck, and M. Chechik. Managing Forked Product Variants. In *Proc. of SPLC'12*, pages 156–160, 2012.

[37] F. Servant and J. A. Jones. History slicing. In *Proc. of ASE'11*, pages 452–455, 2011.

[38] F. Servant and J. A. Jones. History Slicing: Assisting Code-evolution Tasks. In *Proc. of FSE'12*, pages 43:1–43:11, 2012.

[39] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a Java Bytecode Optimization Framework. In *Proc. of CASCON'99*. IBM Press, 1999.

[40] T. Xie and D. Notkin. Checking Inside the Black Box: Regression Testing by Comparing Value Spectra. *IEEE Trans. Softw. Eng.*, 31(10):869–883, Oct. 2005.

[41] A. Zeller. Yesterday, My Program Worked. Today, It Does Not. Why? In *Proc. of ESEC/FSE-7*, pages 253–267, 1999.

[42] A. Zeller and R. Hildebrandt. Simplifying and Isolating Failure-inducing Input. *IEEE Trans. Softw. Eng.*, 28(2):183–200, 2002.

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

[44] S. Zhang, Z. Gu, Y. Lin, and J. Zhao. Change Impact Analysis for AspectJ Programs. In *Proc. of ICSM'08*, pages 87–96, 2008.

[45] T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller. Mining Version Histories to Guide Software Changes. In *Proc. of ICSE'04*, pages 563–572, 2004.