GenSlice: Generalized Semantic History Slicing

***Abstract*—Semantic history slicing addresses the problem of identifying changes related to a particular high-level functionality from the software change histories. Existing solutions are either imprecise, resulting in larger-than-necessary history slices, or inefficient, taking a long time to execute. In this paper, we develop a generalized history slicing framework, named GenSlice, which overcomes the aforementioned limitations. GenSlice abstracts existing history slicing techniques and change history management operations (such as splitting commits into fine-grained changes) as history transformation operators, making it possible to apply them sequentially in various orders. We study and formally prove properties of various orders of operators and devise a systematic approach for efficiently producing history slices that are optimal for practical purposes. We report on an empirical evaluation of our framework, demonstrating its effectiveness on a set of real-world case studies.**

I. Introduction

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

Semantic history slicing [1], [2], [3] addresses the problem of identifying commits related to a particular high-level *functionality* in software change histories. In a concrete instantiation [1], a functionality is defined in terms of a *set of tests*; a particular functionality is considered to be implemented if its corresponding set of tests passes. A *semantic history slice* (*slice* for short) of the original change history is a set of related commits required for the tests to pass.

The slice can be used for many software maintenance tasks. For instance, a slice for a feature available in one software version (or a branch) can be propagated to another software version. Thus, history slicing provides a much needed alternative to the currently-practiced manual process of identifying relevant changes in software change histories.

Motivation. Two recently proposed history slicing techniques – CSlicer [1] and Definer [3]–offer different trade-offs between *efficiency* (i.e., slicing time) and *precision* (i.e., the size of the resulting history slice). CSlicer first runs the set of tests on the latest version of the software and collects code entities required for the tests to pass. It then uses lightweight program analysis techniques to trace back through the change history and identify all changes that are related to the required entities. CSlicer provides *no formal guaranties* for the size of the final history slice, but it is efficient because the set of tests are executed only once. Unlike CSlicer, Definer provides theoretical guarantees that the final history slice is *1-minimal*, i.e., removing any single commit from the resulting slice would lead to a change history that does not implement the functionality of interest. However, using Definer requires executing the set of tests at least once for each program version in the given change history, which is expensive.

Recent work studied the pros and cons of the existing history slicing techniques and attempted to improve their efficiency and precision [2]. On the one hand, with the goal to improve efficiency, Li et al. [1] proposed to run existing slicing techniques in a *sequence*. Specifically, their method runs the lightweight CSlicer first, followed by the more expensive Definer on a much shorter input history, to speed up the latter without sacrificing the slice-minimality guarantee. On the other hand, with the goal to improve precision, Li et al. [3] proposed to *split* the original commits to file-level changes; the key insight is that the file-level split of commits produces finer-grained change histories that can improve the precision of the existing slicing techniques while the original slicing techniques treated each commit as a monolithic entity.

Although these recent advances showed that precision and efficiency can be improved, they are also limited because (a) sequencing and splitting was applied independently, and it is not obvious how to combine the two, and (b) only a single sequence of length two (running CSlicer first and then Definer) was studied, and it is not clear whether longer sequences or different sequences would lead to more efficient slicing and more precise results.

Framework. In this paper, we propose a *generalized history slicing framework*, named GenSlice, which removes the aforementioned limitations. GenSlice abstracts change history management operations (such as splitting commits into finer-grained changes) and existing history slicing techniques as *history transformation operators*, which can be applied in a sequence. We study and formally prove properties of *various orders* of transformation operators and devise a systematic approach for efficiently producing slices which are optimal for practical purposes.

Fig. 1 gives the overview of the framework and illustrates the *history slicing loop*. A single run of the loop on any given change history results in a sequence of transformation operators, and exploring all orders results in a *slicing tree*. Although this tree is, in theory, infinite, we prove that the optimal result, in terms of the history slice length, can be obtained by one of the seven sequences (with no more than five transformation operators in any of these sequences), assuming that Definer always obtains minimal slices, i.e., removing any number of commits from the slices would result in test failures. We study the runtime performance of each such sequence, identifying

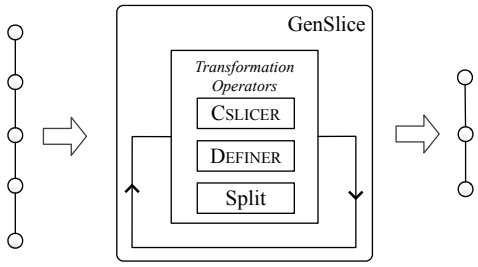

Fig. 1. Overview of GenSlice and the history slicing loop.

those that produce the optimal results in the shortest time.

Contributions. We summarize our contributions as follows:

- **Insight.** We propose a novel abstraction over change history management operations and existing history slicing techniques as transformation operators; these transformation operators can be applied sequentially in various orders to balance efficiency and precision of history slicing.
- **Framework.** We present a framework, GenSlice, which integrates the transformation operators and enables slicing of a given change history in a slicing loop.
- **Proof.** We prove that the optimal result, in terms of the size of the history slice, can be obtained by one of the seven finite sequences of operators, assuming that DEFINER computes minimal slices.
- **Evaluation.** We perform an extensive evaluation to study efficiency of the seven sequences that produce optimal results, applying the sequences on dozens of real-world examples available in open-source projects, which were used in prior work on history slicing. Our results highlight the sequences of operators that are the most efficient in practice.

Organization. The rest of the paper is structured as follows. Sec. II presents an overview of and motivation for GenSlice through an example. Sec. III and IV define terminology and provide the necessary background for semantic history slicing, respectively. Sec. V describes different transformation operators and specifies properties of a slicing tree. In Sec. VI, we describe our implementation and empirical evaluation of the proposed technique. Finally, we discuss related work in Sec. VII and conclude in Sec. VIII.

II. ILLUSTRATIVE EXAMPLE

Consider the brief example, inspired by the functionality csv-159 from the Apache CSV project [4] and shown in Fig. 2. The original change history H is shown horizontally, in the middle of the figure. H consists of five commits: $\langle R1, R2, R3, R4, R5\rangle$. Each commit in the original change history can be split into file-level commits, shown as small rectangles. We distinguish three types of file-level commits: black rectangles are necessary for preserving the functionality; shadowed rectangles are not necessary but are kept by CSLICER due to the imprecision of its lightweight analysis; and white rectangles are irrelevant for the target functionality and are discarded by both CSLICER and DEFINER. We use *C* and *D* to refer to CSLICER and DEFINER when we talk about sequences, respectively, and *S* to refer to the split operator. Consider

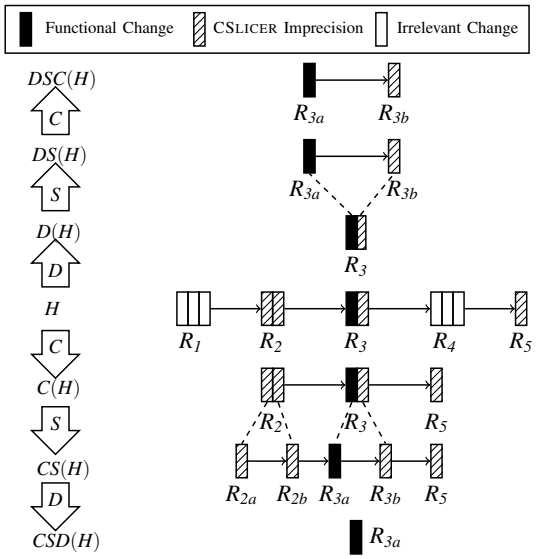

Fig. 2. An example inspired by functionality csv-159 from the Apache CSV project illustrates that both the split operator and the order of applying the operators impact precision of history slicing.

two history slicing sequences: (1) CSLICER→Split→DEFINER (*CSD* for short), and (2) DEFINER→Split→CSLICER (*DSC* for short). Fig. 2 shows step by step application of these two sequences. For example, when running *CSD* (from the middle of the figure moving downwards), after applying *C* on H, the change history becomes $C(H) = \langle R_2, R_3, R_5\rangle$. Then, after applying the *S* operator, $C(H)$ is split into five file-level commits: $CS(H) = \langle R_{2a}, R_{2b}, R_{3a}, R_{3b}, R_5\rangle$. Finally, applying *D* effectively reduces the history to only one file-level commit: $CSD(H) = \langle R_{3a}\rangle$, which is the optimal slice, i.e., the shortest slice that can be obtained by any slicing technique.

The result of slicing, if we switch the order of running CSLICER and DEFINER, is shown in the same figure (from the middle of the figure moving upwards). Interestingly, *DSC* yields a sub-optimal result: $DSC(H) = \langle R_{3a}, R_{3b}\rangle$.

From this example, we can see that the order of applying slicing operators affects the precision of the history slicing. In our example, when applying *D* first, we achieve a much better reduction initially, but fail to exploit the benefit from the fine-grained file-level change histories brought by the split operator, resulting in worse overall results. The reason is that the *D* operator in *DSC* runs on the original commits, and *D* is not able to reduce R_3 which is a mix of necessary and unnecessary changes spreading over multiple files. Thus, even after *D* and *S*, the final *C* operation still could not reduce R_{3b} due to its imprecision.

III. PRELIMINARIES

This section introduces the necessary background and terminology used in the remaining part of the paper.

A. Programs and ASTs

Let P be the syntax rules of a language. We say that a piece of text p is a *syntactically valid* program of language P, denoted by $p \in P$, if p follows the language syntax rules. A

$$\frac{y \in V(r)}{V(r') \leftarrow V(r) \cup \{x\} \quad parent(x) \leftarrow y \quad id(x) \leftarrow n \quad \nu(x) \leftarrow v} \text{INS}((x,n,v),y)$$

$$\frac{x \in V(r)}{V(r') \leftarrow V(r) \setminus \{x\}} \text{DEL}(x) \qquad \frac{x \in V(r)}{\nu(x) \leftarrow v} \text{UPD}(x,v)$$

Fig. 3. The three types of atomic changes as AST transformations [5].

valid program $p \in P$ can be parsed as an *abstract syntax tree* (AST), denoted by $ast(p)$. Formally, $r = ast(p)$ is a rooted tree with a set of nodes $V(r)$. Each 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 and the values are canonical textual representations of the corresponding code entities. We denote the parent of a node x by $parent(x)$.

B. Changes and Change Histories

A change history of a program consists of a sequence of changes, each representing the differences between two versions of the program. These changes can be represented either as differences between ASTs or as unstructured plain text.

1) AST-based View: Let Γ be the set of all ASTs of P. Below, we define changes, change sets, and change histories as a set of AST transformations.

Definition 1 (Atomic change)**.** *An* atomic change $\delta : \Gamma \nrightarrow \Gamma$ *is a partial function which transforms* $r \in \Gamma$*, to produce a new AST* r' *s.t.* $r' = \delta(r)$*. An atomic change can be either an* insert*, a* delete*, or an* update *(see Fig. 3).*

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

Definition 2 (Change set)**.** *Let* r *and* r' *be two ASTs. A* change set $\Delta : \Gamma \nrightarrow \Gamma$ *is a sequence of atomic changes* $\langle \delta_1, \ldots, \delta_n \rangle$*, s.t.* $\Delta(r) = (\delta_n \circ \cdots \circ \delta_1)(r) = r'$*, where* $\circ$ *is function composition.*

A change set $\Delta = \Delta_{-1} \circ \delta_1$ is *applicable* to r if δ_1 is applicable to r and Δ_{-1} is applicable to $\delta_1(r)$. Change sets between two ASTs can be computed by tree differencing [6].

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

Definition 4 (Sub-history)**.** *A* sub-history *is a sub-sequence of a history, i.e., a sequence derived by removing change sets from* H *without altering the ordering.*

We write $H' \subseteq H$ to mean that 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.

2) Text-based View: Software Configuration Management (SCM) tools, e.g., SVN [7] and Git [8], view programs as plain texts and represent program changes using the *text-based view*. The smallest unit for text-based changes is called a *hunk*.

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

Fig. 4. Text-based view of changes represented as a hunk.

Definition 5 (Hunk)**.** *A* hunk $\hat{\delta} : P \nrightarrow P$ *is a partial function which transforms* $p \in P$*, resulting in a new program text* p' *s.t.* $p' = \hat{\delta}(p)$.

For example, Fig. 4 shows a hunk of a one-line deletion and two one-line insertions, marked by "-" and "+", respectively. The *context* (the lines not marked by "-" or "+" in Fig. 4) 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.

A *commit* is a collection of hunks, and a *commit history* is a sequence of commits. Applying a commit is equivalent to applying the composition of the corresponding hunks. More formally,

Definition 6 (Commit)**.** *Let* p *and* p' *be program texts. A* commit $\hat{\Delta} : P \nrightarrow P$ *is a set of hunks* $\{\hat{\delta}_0, \ldots, \hat{\delta}_n\}$ *s.t.* $\hat{\Delta}(p) = (\hat{\delta}_0 \circ \cdots \circ \hat{\delta}_n)(p) = p'$*, where* $\circ$ *is function composition.*

Definition 7 (Commit history)**.** *A* commit history *is a sequence of commits, i.e.,* $H = \langle \hat{\Delta}_1, \ldots, \hat{\Delta}_k \rangle$.

Note that the commit history consisting of commits is a different yet equivalent representation of the change history consisting of atomic changes.

C. Functionality Test

We assume that high level software functionalities such as features and bug fixes can be captured by tests and that the execution trace of a test is deterministic [9]. A *test* t is a predicate $t : P \mapsto \mathbb{B}$, such that for a given program p, $t(p)$ is true if and only if the test passes. A *test suite* is a set of tests that can exercise and demonstrate the functionality of interest. Let a test suite T be a set of tests $\{t_i\}$. We write $p \models T$ if and only if a program p passes T, i.e., $\forall t \in T \cdot t(p)$.

IV. Semantic History Slicing

Semantic history slicing addresses the problem of identifying commits related to a particular high-level functionality from a software change history.

A. Semantics-Preserving History Slices

Consider a program $p_0 \in P$ and its n subsequent versions $p_1, \ldots, p_n$. Let H be the original commit history from p_0 to p_n, i.e., $H_{1..i}(p_0) = p_i$ for all integers $0 \leq i \leq n$. Let T be a test suite passed by p_n, i.e., $p_n \models T$.

Definition 8 (Semantics-preserving slice [1])**.** *A* semantics-preserving slice *of history* H *w.r.t.* T*, denoted by* $H' \subseteq_T H$*, is a sub-history of* H*, i.e.,* $H' \subseteq H$*, s.t.* $H'(p_0) \models T$.

There are several types of semantics-preserving slices: (1) *minimal*, which are semantics-preserving and cannot be reduced further, (2) *optimal*, which are the shortest possible slices of a given history, and (3) *1-minimal*—slices which cannot be further reduced by removing *any single commit*. Clearly, optimal slices are also minimal, but not vice versa. This is because, there might exist a shorter slice H^* which is not a subset of H^m, even if H^m cannot be reduced further. More formally,

Definition 9 (Minimal semantics-preserving slice [1])**.** *A semantics-preserving slice H^m is* minimal *if* $\forall H_{sub} \subset H^m \cdot H_{sub} \not\models T$.

Definition 10 (Optimal semantics-preserving slice)**.** *A semantics-preserving slice H^* is* optimal *if* $\forall H_{sub} \subset H \cdot H_{sub} \models T \implies |H_{sub}| \geq |H^*|$.

Definition 11 (1-minimal semantics-preserving slice)**.** *A semantics-preserving slice H^{1m} is* 1-minimal *if* $\forall H_{sub} \subset H^{1m} \cdot |H_{sub}| = |H^{1m}| - 1 \implies H_{sub} \not\models T$.

While 1-minimal slices are approximate, they are much less expensive to compute than minimal or optimal [1], and in practice are often minimal [3].

B. Computing Semantics-Preserving Slices

With the presence of adequate tests for a functionality and the corresponding change history, semantic history slicing uses tests as the slicing criteria to identify commits in the history (i.e., a semantics-preserving slice) that contribute to the implementation of the given functionality. Two history slicing techniques have been proposed so far, namely, CSLICER, based on change dependency analysis [1] and DEFINER, based on dynamic delta refinement [3].

CSLICER. CSLICER relies on static analysis of dependencies between atomic changes to decide which commits to keep in the history slices. It first analyzes the latest program version to collect test coverage information and then computes an over-approximated set of atomic changes touching the covered elements. Then, through change dependency analysis, it includes additional changes required for proper compilation of the program. Finally, the identified atomic changes are mapped back to the commits in the original change history. The produced history slices are guaranteed to apply without causing any merge conflict, and the resulting program is guaranteed to compile and pass the tests. CSLICER trades precision for efficiency: it conservatively assumes that all changes traversed by the test execution can potentially alter the test results. This assumption results in a possible inclusion of unnecessary or irrelevant changes into the history slice.

DEFINER. In contrast, DEFINER executes tests multiple times and directly observes the test results while attempting to shorten the history slices iteratively. DEFINER derives a 1-minimal semantic slice through the more expensive repeated test executions in a divide-and-conquer fashion that is very similar to *delta debugging* [10]. The high-level idea is to partition the input history by dropping some subset of the commits and opportunistically reduce the search space when the target tests pass on one of the partitions. This process repeats until a 1-minimal partition is reached.

Precision Metrics. History slices of the same length may not carry the same amount of atomic changes or hunks. To better compare the *precision* of different history slicing techniques, we define the *size of a history slice* as the number of changed (added/removed) lines of code in the slice. The *reduction rate* of a history slicing technique is computed as the size of the resulting slice H' divided by the original history size $|H|$, i.e., $|H'|/|H|$. We use this precision metric in the evaluation of all slicing techniques.

V. GENERALIZED HISTORY SLICING

In this section, we describe a generalized history slicing framework which takes a change history as input and applies a sequence of history transformation operators on it until the stopping conditions are met. This framework encompasses all previously studied history slicing techniques and provides an additional flexibility in adjusting the granularity of the change histories. Users are able to configure the slicing technique according to their needs by choosing the right transformation operators and arranging the order in which they are applied.

A. History Transformation Operators

We define two types of *history transformation operators*: *splitting* and *slicing*. Let $H = \langle \Delta_1, \ldots, \Delta_k \rangle$ be a change history. The *split operator* splits a commit into smaller ones, each containing one or more hunks. After the split, the change history becomes more fine-grained.

Definition 12 (Split operator [3])**.** *Let Δ be a commit containing a set of hunks. The* split operator S *partitions Δ s.t.* $\mathrm{S}(\Delta) = \{\Delta'_1, \ldots, \Delta'_n\}$*, where $\Delta'_1, \ldots, \Delta'_n$ are the new fine-grained commits.*

Typically, there is more than one way to split a commit, and in practice, the *file-level split operator* [3], S_{file}, is shown to be the most suitable choice when working with language-agnostic SCM tools. A file-level split operator partitions a commit according to the modified files – the hunks touching the same file are grouped in the same fine-grained commit. Applying a split operator on a change history means simply applying it, in sequence, on each of the commits within the history: $S(H) = \langle S(\Delta_1), \ldots, S(\Delta_k) \rangle$.

The split operator only modifies how changes are grouped, and do not affect the overall effects of change histories. We say that two change histories, H and H', are *equivalent* if, for any program p on which they are applicable, they produce the same new program versions, i.e., $H(p) \equiv H'(p)$. We say that two transformation operators are equivalent if, for any input change history, the histories produced by them are equivalent. For instance, the split operator has the following properties:

$$S(H) \equiv H \qquad \text{(Prop. 1)}$$

$$S(S(H)) \equiv S(H). \qquad \text{(Prop. 2)}$$

Even though the split operator preserves the effects of change histories, it may affect the final results when applied in combination with the slicing operator.

Definition 13 (Slicing operator)**.** *A* slicing operator *SL produces a sub-history s.t.* $SL(H) \subseteq H$.

In this paper, we consider two specific slicing operators, namely, CSLICER (*C*) and DEFINER (*D*), which produce semantics-preserving sub-histories defined by some functionality tests T. None of them promises an optimal solution because computing optimal history slices requires, in the worst-case, enumerating all possible sub-histories [2]. In particular, CSLICER makes no guarantee on the quality of the produced history slices, and repeated applications of CSLICER do not improve the slicing results; DEFINER guarantees that the history slices it computes are 1-minimal (see Sec. IV-B), which may be sub-optimal. Therefore, subsequent applications of DEFINER may result in smaller slices. More formally, the following properties hold:[1]

$$C(C(H)) \equiv C(H) \quad \text{(Prop. 3)}$$
$$D(D(H)) \subseteq D(H), \quad \text{(Prop. 4)}$$

For example, the composition of two consecutive CSLICER calls is equivalent to a single call. In contrast, a subsequent DEFINER call may improve upon the previous result and produce a smaller history slice.

Generally speaking, the slicing operators can produce smaller history slices on more fine-grained inputs. This is always true for CSLICER because its algorithm operates on the atomic changes, and the imprecision is introduced while mapping atomic changes back to coarse-grained commits:

$$C(S(H)) \subseteq C(H) \quad \text{(Prop. 5)}$$
$$C(S(H)) \equiv C(S(C(H))). \quad \text{(Prop. 6)}$$

This also means that running CSLICER before and after the split has the same overall effect as running it only after the split (Prop. 6).

B. Sequences of Transformation Operators

Different history transformation operators can be concatenated in a sequence to construct a *transformation sequence*. For example, passing the result produced by CSLICER to DEFINER gives us $D(C(H))$, shorthanded as $CD(H)$. This transformation sequence is intuitive because CSLICER reduces DEFINER's input size with a relatively small cost, so that running DEFINER later in the sequence is likely faster than standalone. There is an infinite number of transformation sequences, some of which are more efficient than others and some produce shorter history slices. To compare different sequences and identify methodologies to optimize the history slicing results, we systematically study the properties of different sequences.

Slicing tree. Let $\mathcal{K}$ denote the set of all transformation sequences which can be represented as a *slicing tree*—see Fig. 5. The *root* of the tree is the input history H, and every other node represents a sub-history of H. Each node in the tree has n child nodes, where n is the number of possible transformation operators that can be applied on that node. A child node represents the result of applying the corresponding transformation operator. To illustrate, consider the splitting operator *S*, the lightweight slicing operator *C* and the iterative slicing operator *D*. The node *S* has three children: *SS*, *SC*, and *SD*, representing $S(S(H))$, $C(S(H))$, and $D(S(H))$, respectively. There are a few notable facts of the slicing tree. First, its height is unbounded: For any given node, the tree can always be expanded by applying additional transformation operators. Second, all child nodes produce shorter- or equal-length history slices than their parents, by the definitions of the transformation operators. Finally, for a given node in the tree, any sequence that ends with the node requires more computing resources to execute than a sequence ending in the parent of the node.

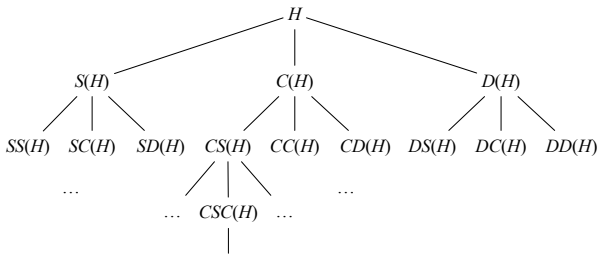

Fig. 5. An example slicing tree over transformation operators *S*, *C*, and *D*.

Navigating the slicing tree. Consider a slicing tree $\mathcal{K}$ defined by three history transformation operators, *S*, *C*, and *D*. Because all three operators preserve the semantic properties, every node in $\mathcal{K}$ represents a semantic history slice of the root. Now the problem of finding a best sequence of history transformations can be converted into the problem of identifying a node in the slicing tree s.t. there exists no semantic history slice shorter than the one produced at that node. Given resource constraints, the slicing tree can be expanded to a limited depth, and the best transformation sequences (of a given length) can thus be derived by comparing all the leaf nodes in a finite tree expansion of the given depth. Here we only optimize over the precision of the transformation sequence, rather than its performance. Later, we study the performance of these sequences empirically in Sec. VI-D.

To efficiently compute all history slices for the leaf nodes (up to some depth of the tree), we propose a tree expansion algorithm with optimizations on *suffix sharing* of the transformation sequences. At the high level, we perform a depth-first-search of the slicing tree to a given depth, and at the same time, cache the intermediate results of explored paths, in case they can be reused in exploring other paths. Alg. 1 shows the details. The inputs to the algorithm are the slicing tree ($\mathcal{K}$), the input change history (H), and the set of tests (T) that defines a functionality of interest. The algorithm returns *slice_map* which maps each sequence k to its corresponding semantic history slice obtained by running the sequence, i.e., $k(H) \models T$. The function EXPLORETREE

[1]The detailed proof of several properties is found in the supplementary materials.

Algorithm 1: Slicing Tree Expansion with Suffix Sharing.

input : $\mathcal{K}$: the slicing tree; H: the input history; T: the functionality test set; *state_map*: the state cache;
output : *slice_map*: a map containing semantic history slice of all the transformation sequences; each key is the name of a sequence, of which the corresponding value is the semantic history slice of H w.r.t. T obtained by running the sequence;

```
1  suffix_map ← ∅;
2  root ← the root of K;
3  slice_map ← EXPLORETREE(K, root, slice_map, H, T, state_map);
4  return slice_map;
5
6  Function EXPLORETREE(K, k, slice_map, H, T, state_map):
7      H' ← RUNSINGLESEQUENCE(k, H, T, state_map);
8      slice_map(k) ← H';
9      foreach k' ∈ the children of k in K do
10         if k' ∉ slice_map then
11             slice_map ←
                 EXPLORETREE(K, k', slice_map, H, T, state_map);
12     return slice_map;
13 Function RUNSINGLESEQUENCE(k, H, T, state_map):
14     if k ∈ state_map then
15         if state_match(H, state_map(k)) then
16             saver_conf ← state_match(H, state_map(k));
17             suffix_map(k)(k) ← suffix_map(k)(matched);
18             return suffix_map(k)(saver_conf);
19     H' ← H;
20     state_map(k)(k) ← H;
21     for m ∈ [1, |H| − 1] do
22         H' ← RUN(X, T, H');
23         if k[m + 1 : |H|] ∈ state_map then
24             if state_match(H', state_map(k[m + 1 : |H|])) then
25                 saver_conf ←
                     state_match(H', state_map(k[m + 1 : |H|]));
26                 suffix_map(k[m + 1 : |H|])(k) ←
                     suffix_map(k[m + 1 : |H|])(saver_conf);
27                 return suffix_map(k[m + 1 : |H|])
28         state_map(k[m + 1 : |H|])(k) ← H';
29     return H';
```

(Lines 6–12) drives the tree exploration, and for each tree node, it calls the RUNSINGLESEQUENCE (Lines 13–29) to execute the transformation sequence and collect the results.

The function RUNSINGLESEQUENCE executes a given transformation sequence k, and naturally reuses the results of any prefix of k which is already executed in previous runs. For example, when running *CD*, the result of *C* (obtained in earlier runs) can be used as the starting point for *D*. In addition, it opportunistically matches the remaining suffix of k with cached runs, in case their results can be reused (Lines 14–18). For example, the sequences *CDSD* and *DSD* share a common suffix *SD*. If their intermediate results after *CD* and *D*, respectively, are exactly the same, then we can be sure that the results of *CDSD* and *DSD* are equivalent as well.

C. Sequence Pruning with Sequence Equivalence

So far, we assumed that the iterative history slicing operator produces 1-minimal results. Our experience shows that in practice, DEFINER produces *minimal* history slices. As computing minimal slices is NP-hard, we cannot validate this hypothesis

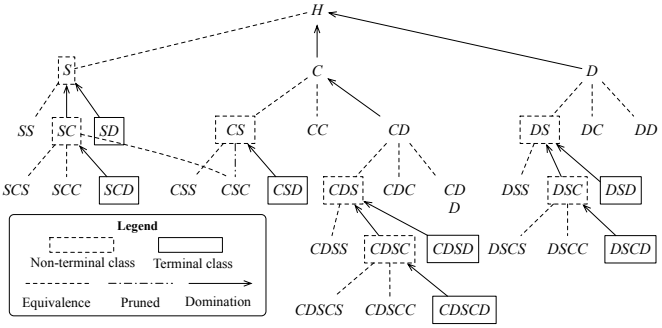

Fig. 6. The slicing tree when *D* is minimal.

for all possible histories. However, we experimented with nine instances (see Tbl. I) where we could fully enumerate all sub-histories of the produced history slices. The results show that none of the slice sub-histories are themselves valid semantics-preserving slices, which means that the results produced by DEFINER are minimal.

While this study does not prove the optimality or minimality of DEFINER, it does suggest potential heuristics which can be used to further optimize the exploration of the slicing tree. In order to have a systematic understanding of the transformation sequences, we study the relationships between different sequences, under two assumptions: (1) when *D* is optimal, and (2) when *D* is minimal.

First, we consider the case when *D* produces *optimal* semantic history slices, i.e., any semantic history slice of H is larger or equal to $D(H)$. Then the best transformation sequence is *SD* (i.e., Split→DEFINER) among all possible sequences ($\mathcal{K}$), as given by Thm. 1:

Theorem 1. *If D produces optimal semantics-preserving slices, then SD is an optimal transformation sequence of* $\mathcal{K}$.

To see this, for any path in $\mathcal{K}$, we consider two cases: (1) *S* is on the path; and (2) *S* is not on the path. For case (1), *SD* is trivially the best transformation sequence, since *D* is optimal and no additional transformation would improve the result. For case (2), suppose there exists a transformation sequence producing a sub-history H' which is smaller than $SD(H)$. Then $S(H')$ is also smaller than $SD(H)$, because of Prop. 1. This contradicts the assumption that *D* is optimal.

Next, we relax the assumption on *D* and consider the case when it produces *minimal* (rather than optimal) semantic history slices, i.e., $D(H)$ cannot be reduced further and stays a valid semantic history slice of H. We rely on Prop. 1 to Prop. 6 when studying the transformation sequences. We say that two transformation sequences are *equivalent* if they always produce the same output histories given the same input histories. A sequence $\mathcal{K}_A$ *dominates* a sequence $\mathcal{K}_B$ if history slices produced by $\mathcal{K}_A$ are always sub-histories of the ones produced by $\mathcal{K}_B$.

Fig. 6 shows the slicing tree, assuming *D* is minimal. The solid arrows between tree nodes indicate the dominance relations. For instance, the arrow pointing from *CD* to *C* indicates that *C* is dominated by *CD* since the latter always produces a sub-history of the former. The dashed lines between

two nodes indicate that the two corresponding transformation sequences are equivalent. For example, *S* and *SS* are connected by a dashed line since we have $S(H) \equiv SS(H)$ (recall Prop. 2). Based on the equivalence relations, some sequences can be pruned because it suffices to run only one of the equivalent sequences. For instance, *CSC* can be pruned since it is equivalent to *SC* (given by Prop. 6). Note that any expansion to the tree in Fig. 6 results in a sequence equivalent to one of the existing tree node, i.e., all equivalence classes of $\mathcal{K}$ are captured by this tree.

In particular, we would like to find out whether the set of transformation sequences can be classified into a small number of *equivalence classes* according to their domination power. Based on this, we can then identify the *optimal transformation sequences* which are not dominated by others. Under the assumption that *D* is minimal, we are able to show that the optimal transformation sequences are within at most seven choices.

Theorem 2. *If D produces minimal semantics-preserving slices, then the optimal transformation sequences of $\mathcal{K}$ are within the set $\mathcal{K}^* = \{SD, SCD, CSD, CDSD, CDSCD, DSD, DSCD\}$.*

Proof. We begin by recognizing that there are the total of 14 equivalence classes of $\mathcal{K}$, namely, $EC_{\mathcal{K}} = \{[S], [CS], [DS], [SC], [SD], [CDS], [SCD], [CSD], [DSC], [DSD], [CDSC], [CDSD], [DSCD], [CDSCD]\}$, as highlighted in Fig. 6. We use $[X]$ to denote the *equivalence class* of transformation sequences represented by *X*. The nodes surrounded by dashed boxes are *non-terminal classes* and the ones surrounded by solid boxes are *terminal classes*. We prove by induction that any other transformation sequence $k \in \mathcal{K}$ belongs to one of these equivalence classes.

First, consider the base case where $k_0 \in \{C, D, CD\}$ or $[k_0] \in EC_{\mathcal{K}}$. Since $C \equiv CS$, $D \equiv DS$, and $CD \equiv CDS$ by Prop. 2, $[k_0] \in EC_{\mathcal{K}}$ and the induction hypothesis trivially holds. Next, for any transformation sequence k, suppose $[k] \in EC_{\mathcal{K}}$ holds. We need to show that if any one of the three transformation operators (*S*, *C*, and *D*) is appended to k, the resulting transformation sequence k' still belongs to the equivalence classes in $EC_{\mathcal{K}}$, i.e., $k' \in [k]$. This can be proven by considering all 14 equivalence classes case-by-case. We do one of these; others are very similar. When $k \in [CS]$, k' is equivalent to either *CSS*, *CSC*, or *CSD*. Again, we have $CSS \equiv CS$ by Prop. 2; hence $CSS \in [CS]$. We also have $CSC \equiv SC$ by Prop. 6, and thus $CSC \in [SC]$. Finally, *CSD* belongs to $[CSD]$; therefore, the hypothesis holds when $k \in [CS]$.

With the set of equivalence classes established, we then show that the optimal transformation sequences of $\mathcal{K}$ are within the seven sequences identified by $\mathcal{K}^*$. We can divide $EC_{\mathcal{K}}$ into the *terminal classes* ($\mathcal{K}^*$) and the *non-terminal classes* ($EC_{\mathcal{K}} \setminus \mathcal{K}^*$), and show by case analysis that every non-terminal class is dominated by some terminal class by Def. 13, i.e., $D(H) \subseteq H$ for any H. Therefore, the optimal transformation sequence not dominated by any other sequences must be one of the terminal classes $\mathcal{K}^*$. □

GenSlice. Based on Thm. 2, a practical and efficient way to obtain the optimal transformation sequence is to explore the pruned slicing tree following Alg. 1 and compare the seven terminal nodes. The best results among the seven are also the best achievable (optimal) history slices with any transformation sequence, given that DEFINER is minimal. We empirically evaluate the precision and efficiency of GenSlice in Sec. VI-C and VI-D, respectively.

VI. EVALUATION

We aim to answer the following questions through the empirical evaluation:

RQ1 (Applicability): How frequently does the assumption that *D* is minimal hold in practice?

RQ2 (Precision): What is the improvement on precision of optimal slicing transformation sequence(s) compared with other transformation sequences?

RQ3 (Efficiency): Which optimal slicing transformation sequence(s) are the most efficient?

RQ1 aims to find out whether GenSlice's theoretically optimal sequences given by Thm. 2 are applicable in practice; RQ2 aims to measure the precision improvement of the optimal transformation sequences over non-optimal transformation sequences. RQ3 aims to compare the efficiency among the optimal sequences and identify the most efficient ones.

A. Subjects

We conducted the experiments on a benchmark consisting of 28 functionalities selected from the DoSC dataset [11]. Each functionality is identified by a unique key which refers to its corresponding issue ID on the JIRA issue tracker [12] and is accompanied by a set of tests. DoSC includes the starting and ending versions of the software history which determine the development life cycle of a functionality.

DoSC includes functionalities from nine open source projects: Compress [13], Configuration [14], CSV [4], Flume [15], IO [16], Lang [17], Maven [18], Net [19], and PDFBox [20]. All of these projects are written in Java and have openly accessible change histories. Initially, five functionalities were randomly selected from each project available in DoSC. We then removed functionalities that are currently not supported by CSLICER (e.g., those which include changes modifying non-Java files) or take more than two hours to analyze with DEFINER. In the end, our experiments used 28 functionalities from nine projects—see Fig. 9 for their unique keys.

We conducted the experiments on a 6-core Intel(R) Core(TM) i7-8700 CPU@3.20 GHz machine with 64GB of RAM, running Ubuntu 18.04, with Java 1.8.0_151 and Python 3.6.7.

B. RQ1: Applicability

In the first experiment, we enumerated the history slices obtained by running the standalone *D* on the subjects, checking whether any subset of $D(H)$ is still a functionality-preserving history slice. Note that not all the results of $D(H)$ of all the subjects are feasible to enumerate. A given $D(H)$ with N commits requires running 2^N combinations to check its

TABLE I
THE RESULT OF MINIMALITY CHECKING OF D IN PRACTICE.

Functionality	$\|H\|$	$D(H)$	Minimal?
compress-375	15716	110	Yes
configuration-626	3170	45	Yes
csv-159	3489	136	Yes
flume-2628	28542	392	Yes
io-275	11153	203	Yes
io-288	11153	842	Yes
pdfbox-3069	16393	478	Yes
pdfbox-3307	1440	76	Yes
pdfbox-3418	16393	331	Yes

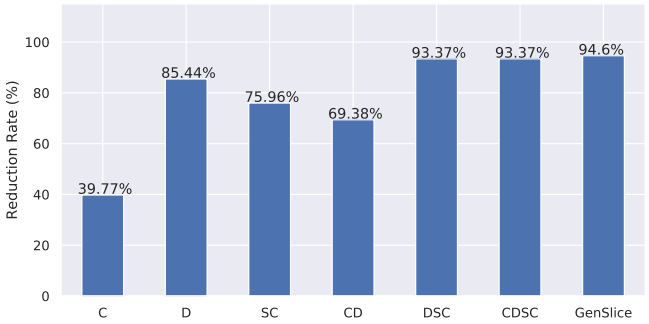

Fig. 7. Comparison of the average reduction rate of the sequences in the slicing tree. (The bar "GenSlice" represents the (same) reduction rate obtained by the seven theoretically optimal sequences proved in Thm. 2.)

minimality. Each combination includes cherry-picking all of the commits in the history slice to a new branch and executing tests to check if they pass. As N increases, the resources required by enumeration grow exponentially. Thus, we set a limit $N = 5$, and only performed the minimality checking on the subjects where $D(H) < N$. As a result, we performed minimality checking on nine subjects. These subjects are from six different projects, which range from simple IO handling to large distributed services. Structure-wise, there are four single-module and two multi-module Maven projects. We consider these subjects reasonably broad and diverse for our experiment.

Table I shows the results. Column $|H|$ represents the size of the original history. Column $D(H)$ represents the size of the history slice obtained by D. The last column shows whether $D(H)$ is the minimal history slice. The results indicate that D obtained the minimal history slice on all of the functionalities where we performed the minimality checking.

Answer to RQ1. On all the functionalities where $D(H)$ is feasible to enumerate, the assumption that D is minimal holds. This finding supports our claim that GenSlice's theoretically optimal sequences defined in Thm. 2 are applicable in practice.

C. RQ2: Precision

Our second experiment involves running all of the sequences of the non-terminal classes in Fig. 6, comparing their reduction rate with the result of GenSlice, and measuring the improvement of the optimal slicing sequences over other sequences. Fig. 7 shows the result of the comparison; each bar depicts the average reduction rate of a sequence over the original history, in terms of the number of changed lines, on all the subjects. The bars are sorted in the order of levels of the tree in Fig. 6. The rightmost bar, "GenSlice", represents the reduction rate of the optimal sequences, 94.60%, over the original history. We did not compare with S (splitting standalone) because S changes only the granularity of commits but does not perform reduction. Similarly, $C \equiv CS$, $D \equiv DS$, and $CD \equiv CDS$ in terms of the slice size. Thus, we only compare with C, D, and CD, but not with CS, DS, or CDS. The maximum improvement of GenSlice over other sequences is observed w.r.t. the sequence C (137.87%). The minimum improvement of GenSlice is observed w.r.t. sequences DSC and $CDSC$ (1.32%). On average, across all non-terminal class sequences, the improvement of GenSlice is 35.35%.

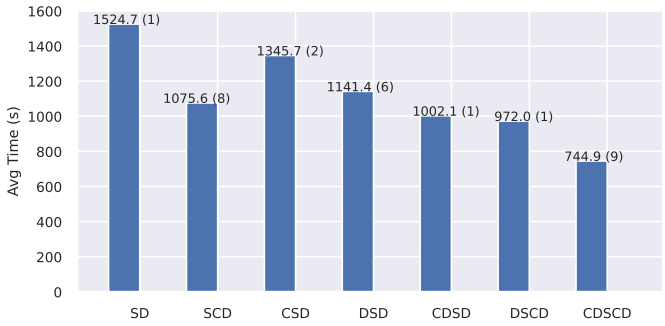

Fig. 8. Comparison of the execution time of the optimal sequences in Thm. 2.

Answer to RQ2. On average, GenSlice obtains a 35.35% improvement in the reduction rate over all non-terminal class sequences, with the maximum being 137.87% and the minimum being 1.32%. The results confirm that the theoretically optimal sequences of GenSlice are more precise than others.

D. RQ3: Efficiency

Next, we compare the execution time across all the optimal sequences—see Fig. 8 for the results. Blue bars show the execution time of each sequence, averaging over all the 28 subjects; the number of subjects on which each sequence achieves the best performance is shown in the parentheses of the annotation above each bar. We ran each sequence five times for each functionality and report the average. For each sequence, the average variation across the executions is always less than 3%. The detailed timing data is available at: https://sites.google.com/view/genslice.

Based on these results, we make the following observations. First, of all the sequences, *CDSCD* achieves the best efficiency overall. It takes the least time, on average, and obtains the best efficiency on more subjects. Second, *SD* performs the worst overall, taking the longest time, on average, and performing the best on only one case.

SD is the least efficient sequence because S increases the number of commits, bringing more workload on D. We use a case analysis of io-305 to show how adding a certain operator, e.g., C, speeds up *SD*'s execution. We choose this case because on it, *SCD* (427.76 seconds) is significantly (1.8X) faster than *SD* (770.22 seconds). For both sequences, the first S splits the original history into 546 file-level commits. After that, *SD*'s D

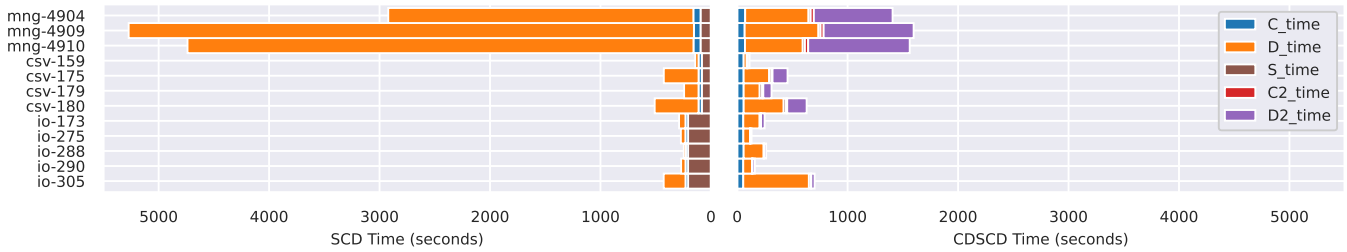

Fig. 9. Comparison of the execution time of the transformation operators: SCD vs CDSCD.

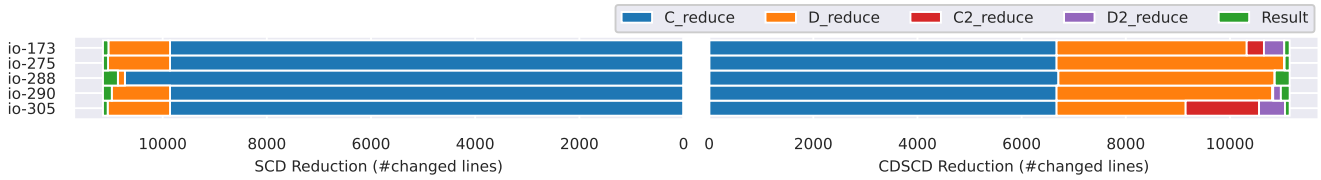

Fig. 10. Comparison of the line of changes reduced by transformation operators: SCD vs CDSCD.

phase is directly executed on all of these file-level commits. On the other hand, *SCD*'s *C* phase significantly reduces the history slice, from 546 to 52 file-level commits (taking only 23 seconds), greatly decreasing the workload of the subsequent *D* phase and thus resulting in a substantial speedup.

To study the difference in execution time between different sequences, we detailed out the execution time and the reduction rate of each operator in a sequence. Fig. 9 shows the execution times and Figure 10 show the reduction rates for each operator in two sequences, *CDSCD* and *SCD*, on an illustrative subset of the subjects. Similar results for all of the subjects are available on our website.

We observe that *CDSCD* performs well for projects with *focused* commit, i.e., when each commit in the selected history range either directly contributes to the target functionality or is irrelevant to it. The three mng examples (mng-4904, mng-4909, mng-4910) in Fig. 9 are of this kind: the commits in their final history slice contribute to the target functionality in full. Splitting such commits is ineffective as GenSlice always makes the same decisions on all commit parts. Therefore, *S* has little effect on the slicing result, i.e., *SCD* has almost the same precision as *CD*. On the other hand, as *S* increases the number of commits for the subsequent *CD* to process, it decreases the overall efficiency of the sequence. We believe that creating focused commits is a good practice, advocated by many developers [21], [22], as it simplifies maintenance and collaboration. The fact that all the three functionalities belong to the Maven project also indicates that Maven developers are in the habit of creating focused commits.

Our second finding is that the size of the target functionality also influences the efficiency of some transformation sequences. This is because, the lengths of the history slices produced by the *C* operator grows when the target functionality is large (its corresponding test cases cover many code entities). In general, on large functionalities, *CDSCD* clearly outperforms *SCD*; otherwise, their difference is small. More specifically, the four csv examples (csv-159, csv-175, csv-179, and csv-180) in Fig. 9 are also considered focused, but *CDSCD* and *SCD* have similar performance. *SCD* even takes a slightly shorter time on three of these cases. The reason is that these csv functionalities are much smaller than those of the mng examples. Thus, although their original histories are of similar lengths, after *SC*, *D* needs to process only 38 file-level commits for csv (it varies from 77 to 82 for mng), making its execution fast and its efficiency similar to that of *CDSCD*.

Our third finding is that if the 1-minimal history slice is very small, then running *S* first is potentially wasteful. For example, *S* takes most of the time during executing *SCD* on io-173, io-275, io-288, and io-290 (see Fig. 9), causing this sequence to be less efficient than *CDSCD*. The reason is that *D*'s underlying technique is a delta debugging-style refinement. If the 1-minimal result is small, it can quickly drop a large portion of the commits. For such cases, running *CD* first is a more appropriate choice.

Our final finding is that if the commits on the target functionality are not focused, *D* tends to be inefficient. For example, the first *D* in *CDSCD* spends much more time on io-305 than on the other four io examples. Combining with results in Figs. 9 and 10, we also observed that running *D* on io-305 reduces much less changes than on other io examples. Inspecting io-305 manually, we determined that its functionality is not focused but rather implemented by multiple commits spreading over the history, where each relevant commit has multiple purposes. *D*'s partition is extremely inefficient in such a case, incurring the large overhead.

Interestingly, in practice, longer sequences of operators may be more efficient than shorter ones. For instance, *SCD* and *CDSCD* both outperformed *SD* in terms of the execution time. *CDSCD* suggests a methodology of pipelining transformation operators: we should begin by applying coarse-grained analysis to obtain an initial valid history slice and then apply fine-grained analysis only on those commits that need further refinement.

Answer to RQ3. Of the seven sequences in Thm. 2, *CDSCD* is deemed to be the best overall: its achieves the least average execution time among all optimal sequences. Therefore, we recommend that developers use *CDSCD* in history slicing tasks. We also discover two important characteristics—focusing and size—of the target functionality that influence the efficiency of

sequences. *CDSCD* performs extremely well when the target functionality is focused and large.

E. Threats to Validity

Our work has several threats to validity. Our findings may not generalize to software projects other than those used in our evaluation. To mitigate this threat, we used several projects from the largest dataset related to history slicing.

Results reported in this paper were obtained on a single machine, and our findings related to execution time might differ on another machine. While working on the implementation of our tool, we have used several machines with different hardware configurations and observed similar trends in terms of execution time. We also repeated each experiment five times and averaged the execution times.

Our scripts for running experiments and our implementation of the slicing operators may have bugs. We mitigate this threat by implementing our tool on top of existing history slicing tools. We have also checked for potential outliers in our results. We will make our code and scripts available for reproducability.

VII. Related Work

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 [23], [24], [25], [26], localize bugs [10], [27], and suggest likely further code changes [28], [29].

A. Change History Transformations

Servant and Jones [26] first proposed the concept of history slicing, which extracts version histories relevant to a selected set of lines of code. The results of history slicing is a graph that links every line of the selected code with its corresponding previous version through the history of the software project. The goal is to provide a reduced amount of information about the code of interest, in order to assist a number of tasks such as inferring design rationale from past code changes or assessing developer expertise for a software feature or bug. However, *history slicing* [26] and *semantic history slicing* [1] are two different techniques. The former operates completely on the syntactic level, i.e., on the text-based view of change histories. The latter takes into account both the syntax and the semantics of changes, and produces semantics-preserving history slices.

Another interesting take on history analysis is *history transformation* [30], [23]. Muşlu et al. [23] 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 change histories into different representations at various levels of granularity to better facilitate the tasks at hand. As a potential future direction, such transformations can be combined with semantic history slicing to build a view of change history that clusters semantically related changes. Barnett et al. [31] proposed an automatic technique for decomposing change sets into multiple independent code differences based on syntactical relationships between changes, aiming to assist developers to understand changes in a code review. Unlike their approach, GenSlice splits commits based on modified files, and our goal of splitting is to improve the precision of semantic history slicing techniques.

B. Fault Localization in Version Histories

Delta debugging [10] uses divide-and-conquer-style iterative test executions to narrow down the causes of software failures. Delta debugging was applied to minimize the set of changes which cause regression test failures. This problem can be considered as finding minimal semantic history slices w.r.t. the failure-inducing properties. However, in contrast to DEFINER (one of the operators in GenSlice), 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 [27] consider an approximated version of minimality, i.e., 1-minimality, which guarantees that removing any single change breaks the target properties.

Selective bisection debugging [32] is an enhancement over the traditional *bisection debugging* approaches based on binary search over a software change history (e.g., Git-bisect [33]). Bisection debugging is widely used in practice to identify bug introducing commits, but it can be expensive due to costly compilation and test execution. 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 a failing test and thus can be skipped to save time. This is similar to DEFINER as they both rely on test signals to make predictions to reduce overall cost of computation. Conceptually, though, there is a difference: selective bisection debugging tries to find which commits cause the test to fail, while we look for commits that cause the test to pass. Another difference is that bisection debugging tries to locate a certain version while GenSlice attempts to find a 1-minimal history slice, which is a more difficult problem ($\mathcal{O}(\log n)$ vs. $\mathcal{O}(n^2)$).

VIII. Conclusions

Semantic history slicing addresses the problem of identifying changes related to a particular high-level functionality from the software change histories. Existing solutions to this problem are either imprecise, resulting in larger-than-necessary history slices, or inefficient, taking a long time to execute. To overcome these limitations, we presented GenSlice, a novel framework that abstracts change history management operations and existing history slicing techniques as transformation operators, which can be applied sequentially in various orders. We studied and formally proved properties of several operator combinations and devised a systematic approach for efficiently producing history slices which are optimal for practical purposes. We reported on an empirical evaluation of our framework demonstrating its effectiveness on a set of real-world case studies and highlighted the most efficient sequences.

REFERENCES

[1] Y. Li, C. Zhu, J. Rubin, and M. Chechik, "Semantic Slicing of Software Version Histories," *IEEE Transactions on Software Engineering*, vol. 44, no. 2, pp. 182–201, 2017.

[2] Y. Li, "Managing Software Evolution Through Semantic History Slicing," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, 2017, pp. 1014–1017.

[3] Y. Li, C. Zhu, M. Gligoric, J. Rubin, and M. Chechik, "Precise semantic history slicing through dynamic delta refinement," *Automated Software Engineering*, vol. 26, no. 4, pp. 757–793, 2019.

[4] "Commons CSV Library," https://commons.apache.org/proper/commons-csv, 2019.

[5] B. Fluri and H. C. Gall, "Classifying Change Types for Qualifying Change Couplings," in *Proceedings of the 14th IEEE International Conference on Program Comprehension*, 2006, pp. 35–45.

[6] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom, "Change Detection in Hierarchically Structured Information," in *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, 1996, pp. 493–504.

[7] "Subversion (SVN) Version Control System," http://subversion.apache.org, 2019.

[8] "Git Version Control System," https://git-scm.com, 2019.

[9] G. Rothermel and M. J. Harrold, "Analyzing Regression Test Selection Techniques," *IEEE Transactions on Software Engineering*, vol. 22, no. 8, pp. 529–551, 1996.

[10] A. Zeller, "Yesterday, My Program Worked. Today, It Does Not. Why?" in *Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 1999, pp. 253–267.

[11] C. Zhu, Y. Li, J. Rubin, and M. Chechik, "A Dataset for Dynamic Discovery of Semantic Changes in Version Controlled Software Histories," in *Proceedings of the 14th International Conference on Mining Software Repositories*, 2017, pp. 523–526.

[12] "JIRA Software," https://www.atlassian.com/software/jira, 2019.

[13] "Commons Compress Library," https://commons.apache.org/proper/commons-compress, 2019.

[14] "Commons Configuration Library," https://commons.apache.org/proper/commons-configuration, 2019.

[15] "Flume," https://flume.apache.org, 2019.

[16] "Commons IO Library," https://commons.apache.org/proper/commons-io, 2019.

[17] "Commons Lang Library," https://commons.apache.org/proper/commons-lang, 2019.

[18] "Maven Project," https://maven.apache.org, 2019.

[19] "Commons Net Library," https://commons.apache.org/proper/commons-net, 2019.

[20] "PDFBox – A Java PDF Library," https://pdfbox.apache.org, 2019.

[21] "Best practices for using Git," https://deepsource.io/blog/git-best-practices, 2020.

[22] "Git Best Practices," https://programmerfriend.com/git-best-practices, 2020.

[23] K. Muşlu, L. Swart, Y. Brun, and M. D. Ernst, "Development History Granularity Transformations," in *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, 2015, pp. 697–702.

[24] E. Murphy-Hill, C. Parnin, and A. P. Black, "How We Refactor, and How We Know It," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 5–18, 2012.

[25] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin, "Early Detection of Collaboration Conflicts and Risks," *IEEE Transactions on Software Engineering*, vol. 39, no. 10, pp. 1358–1375, 2013.

[26] F. Servant and J. A. Jones, "History slicing," in *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering*, 2011, pp. 452–455.

[27] A. Zeller and R. Hildebrandt, "Simplifying and Isolating Failure-Inducing Input," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 183–200, 2002.

[28] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller, "Mining Version Histories to Guide Software Changes," in *Proceedings of the 26th International Conference on Software Engineering*, 2004, pp. 563–572.

[30] S. Hayashi, T. Omori, T. Zenmyo, K. Maruyama, and M. Saeki, "Refactoring Edit History of Source Code," in *Proceedings of the 28th IEEE International Conference on Software Maintenance*, 2012, pp. 617–620.

[31] M. Barnett, C. Bird, J. Brunet, and S. K. Lahiri, "Helping developers help themselves: Automatic decomposition of code review changesets," in *Proceedings of the 37th International Conference on Software Engineering*, 2015, pp. 134–144.

[32] R. Saha and M. Gligoric, "Selective Bisection Debugging," in *Proceedings of the 20th International Conference on Fundamental Approaches to Software Engineering*, 2017, pp. 60–77.

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