Verified Three-Way Program Merge

MARCELO SOUSA, University of Oxford, United Kingdom
ISIL DILLIG, University of Texas at Austin, United States
SHUVENDU K. LAHIRI, Microsoft Research, United States

Even though many programmers rely on 3-way merge tools to integrate changes from different branches, such tools can introduce subtle bugs in the integration process. This paper aims to mitigate this problem by defining a semantic notion of *conflict-freedom*, which ensures that the merged program does not introduce new unwanted behaviors. We also show how to verify this property using a novel, compositional algorithm that combines lightweight summarization for shared program fragments with precise relational reasoning for the modifications. Towards this goal, our method uses a 4-way differencing algorithm on abstract syntax trees to represent different program versions as edits applied to a shared program with holes. This representation allows our verification algorithm to reason about different edits in isolation and compose them to obtain an overall proof of conflict freedom. We have implemented the proposed technique in a new tool called SafeMerge for Java and evaluate it on 52 real-world merge scenarios obtained from Github. The experimental results demonstrate the benefits of our approach over syntactic conflict-freedom and indicate that SafeMerge is both precise and practical.

CCS Concepts: • **Software and its engineering → Formal software verification**;

Additional Key Words and Phrases: Three-way program merge, relational verification, product programs

ACM Reference Format:
Marcelo Sousa, Isil Dillig, and Shuvendu K. Lahiri. 2018. Verified Three-Way Program Merge. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 165 (November 2018), 29 pages. https://doi.org/10.1145/3276535

1 INTRODUCTION

Developers who edit different branches of a source code repository rely on 3-way merge tools (like `git-merge` or `kdiff3`) to automatically merge their changes. Since the vast majority of these tools are oblivious to program semantics and resolve conflicts using syntactic criteria, they can —and, in practice, do— introduce bugs in the merge process. For example, numerous on-line posts illustrate the pitfalls of textual merge tools [Fowler, Martin 2011; Lee, TK 2012; Lenski, Dan 2015; Reddit 2017b] and provide ample examples of bugs that are introduced in the merge process [David Wheeler 2017; Knoy, Gabriel 2012; Lutton, Mark 2014; Reddit 2017a; Rostedt, Steven 2011]. Furthermore, according to a simple empirical study that we performed on public Github data, there are 3500 commits that likely introduce a bug during the merge. [1] According to several sources, the infamous

[1] We estimate this number by performing a query over Github public data on commits searching for "*merge.*introduce.*bug" on the subject and description of the commit.

Authors' addresses: Marcelo Sousa, University of Oxford, United Kingdom, marcelo.sousa@cs.ox.ac.uk; Isil Dillig, University of Texas at Austin, United States, isil@cs.utexas.edu; Shuvendu K. Lahiri, Microsoft Research, United States, shuvendu.lahiri@microsoft.com.

2475-1421/2018/11-ART165
https://doi.org/10.1145/3276535

Apple SSL bug (that resulted from duplicate goto statements) may have been introduced in the automatic merge process [David Wheeler 2017; John Gruber 2014; SlashDot 2014].

To see how bugs may be introduced in the merge process, consider the simple *base program* shown in Figure 1 together with its two *variants* A and B. [2] In this example, the *base* program contains a redundant guard against *null* on the return of `malloc`. One can either remove the first check for *null* (as done in variant A) or the second check for non-*null* (as done in variant B). However, the merge generated by running a standard 3-way merge tool (in this case, `git merge` with default options) generates an incorrect merge *merge* that removes *both* the checks, which exposes the null-dereference bug not present in any of the three versions.

Base

```
void f() {
  p = malloc(4);
  if (!p) {
    return;
  }
  q = 2;
  if (p) {
    *p = 1;
  }
}
```

Variant A

```
void f() {
  p = malloc(4);
  q = 2;
  if (p) {
    *p = 1;
  }
}
```

Variant B

```
void f() {
  p = malloc(4);
  if (!p) {
    return;
  }
  q = 2;
  *p = 1;
}
```

Generated Merge

```
void f() {
  p = malloc(4);
  q = 2;
  *p = 1;
}
```

Fig. 1. Simple motivating example.

This paper takes a step towards eliminating bugs that arise due to 3-way program merges by automatically verifying *semantic conflict-freedom*, a notion inspired by earlier work on program integration [Horwitz et al. 1989; Yang et al. 1990]. To motivate what we mean by semantic conflict-freedom, consider a base program P, two variants A, B, and a merge candidate M. Intuitively, semantic conflict freedom requires that, if variant A (resp. B) disagrees with P on the value of some program variable v, then the merge candidate M should agree with A (resp. B) on the value of v. In addition to ensuring that the merge candidate does not introduce new behavior that is not present in either of the variants, conflict freedom also ensures that variants A and B do not make changes that are semantically incompatible with each other.

The main contribution of this paper is a novel compositional verification algorithm, and its implementation in a tool called SafeMerge, for automatically proving semantic conflict-freedom of Java programs. Our method is compositional in that it analyzes different modifications to the program in isolation and composes them to obtain an overall proof of semantic conflict-freedom. A key idea that allows compositionality is to model different versions of the program using *edits* applied to a *shared program with holes*. Specifically, the shared program captures common statements between the program versions, and holes represent discrepancies between them. The edits describe how to fill each hole in the shared program to obtain the corresponding statement in a variant. Given such a representation that is automatically generated by SafeMerge, our verification algorithm uses lightweight analysis to reason about shared program fragments but resorts to precise relational techniques to reason about modifications.

[2] This example is inspired from a presentation by Jim Larus https://barghouthi.github.io/repsatsixty/larus.pdf.

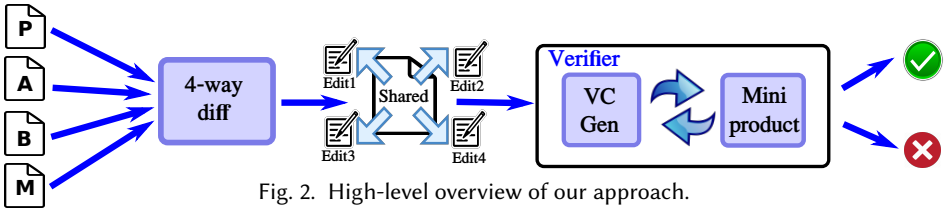

Fig. 2. High-level overview of our approach.

The overall workflow of our approach is illustrated schematically in Figure 2. Our method takes as input four related programs, namely the original program P, two variants A and B, and a merge candidate M, and represents them as edits applied to a shared program by running a "4-way diff" algorithm on the abstract syntax trees. The verifier leverages the result of the 4-way diff algorithm to identify which parts of the program to analyze more precisely. Specifically, our verification algorithm summarizes shared program fragments using uninterpreted functions of the form $x = f(x_1, \ldots, x_n)$ that encode dependencies between program variables. In contrast, the verifier reasons about edited program fragments in a more fine-grained way by constructing 4-way *product programs* that encode the simultaneous behavior of all four edits. Overall, this interplay between lightweight dependence analysis and product construction allows our technique to generate verification conditions whose complexity depends on the size and number of the edits.

To evaluate our technique, we collected over 50 real-world merge scenarios obtained by crawling Github commit histories and evaluate SafeMerge on these benchmarks. Our tool is able to verify the correctness of the merge candidate in 75% of the benchmarks and identifies eleven real violations of semantic conflict-freedom, some of which are not detected by textual merge tools. Our evaluation also demonstrates the scalability of our method and illustrates the advantages of performing compositional reasoning.

In summary, this paper makes the following key contributions:

- We introduce the merge verification problem based on the notion of semantic conflict-freedom.
- We provide a compositional verification algorithm that combines precise relational reasoning about the edits with lightweight reasoning for unedited program fragments.
- We present a novel n-way product construction technique for precise relational verification.
- We describe an n-way AST diff algorithm and use it to represent program versions as edits applied to a shared program with holes.
- We implement our method in a tool called SafeMerge and evaluate our approach on real-world merge scenarios collected from Github repositories.

2 OVERVIEW

In this section, we give an overview of our approach with the aid of a merge example from the `RxJava` project [3], a popular library for composing asynchronous and event-based programs using observable sequences for the Java VM. Figure 3 shows the Base version (O) of the `triggerActions` method from the `TestScheduler.java` file. The two variants $\mathcal{A}$, $\mathcal{B}$ and the merge $\mathcal{M}$ perform the following modifications:

- Variant $\mathcal{A}$ *moves* the statement `time = targetTimeInNanos` at line 7 to immediately after the while loop. This modification potentially impacts the value of the variable `time` in $\mathcal{A}$ with respect to the Base version.
- Variant $\mathcal{B}$ guards the call `current.action.call(...)` at line 12 with the condition:

  ```
  if(!current.isCancelled.get()) {...}
  ```

  The call (at line 12) has a side effect on the variable called `value` which changes the effect on `value` with respect to the Base version.
- The merge $\mathcal{M}$ incorporates both of these changes.

This example is interesting in that both variants modify code within a loop, and one of them (namely, $\mathcal{B}$) changes the control-flow by introducing a conditional. The loop in turn depends on the state of an unbounded collection `queue`, which is manipulated using methods such as `queue.isEmpty`

[3] https://github.com/ReactiveX/RxJava/commit/1c47b0c.

```
1  Queue<TimedAction> queue = new PriorityQueue<TimedAction>(...);
2  int time;  int value;
3  void triggerActions(long targetTimeInNanos) {
4      while(!queue.isEmpty()){
5           TimedAction current = queue.peek();
6           if(current.time > targetTimeInNanos){
7               time = targetTimeInNanos;
8               break;
9           }
10          time = current.time;
11          queue.remove();
12          current.action.call(current.scheduler, current.state);
13     }  }
```

Fig. 3. Procedure from the base program in `RxJava`.

Shared program with holes ($\hat{\mathcal{S}}$)

```
void triggerActions(long targetTimeInNanos) {
    while (!queue.isEmpty()) {
        TimedAction current = queue.peek();
        if (current.time > targetTimeInNanos) {
           <?HOLE?>;
           break; }
        time = current.time; queue.remove();
        <?HOLE?>;
    }
    <?HOLE?>;
}
```

Edit $\mathcal{O}$ ($\Delta_{\mathcal{O}}$)

```
[ time = targetTimeInNanos, current.action.call(...), skip ]
```

Edit $\mathcal{A}$ ($\Delta_{\mathcal{A}}$)

```
[ skip, current.action.call(...), time = targetTimeInNanos ]
```

Edit $\mathcal{B}$ ($\Delta_{\mathcal{B}}$)

```
[ time = targetTimeInNanos,
  if(!current.isCancelled.get()) { current.action.call(...);},
  skip ]
```

Edit $\mathcal{M}$ ($\Delta_{\mathcal{M}}$)

```
[ skip,
  if(!current.isCancelled.get()) { current.action.call(...); },
  time = targetTimeInNanos ]
```

Fig. 4. Shared program with holes and the edits.

and `queue.remove`. Furthermore, while `triggerActions` has no return value, it has implicit side-effects on variables `time` and `value`, and on the collection `queue`. Together, these features make it challenging to ensure that the merge $\mathcal{M}$ preserves changes from both variants and does not introduce any new behavior.

To verify semantic conflict-freedom, our technique represents the changes formally using a list of *edits* over a shared program with *holes*. Figure 4 shows the shared program $\hat{\mathcal{S}}$ along with the corresponding edits $\Delta_{\mathcal{O}}, \Delta_{\mathcal{A}}, \Delta_{\mathcal{B}}, \Delta_{\mathcal{M}}$. A hole (denoted as <?HOLE?>) in $\hat{\mathcal{S}}$ is a placeholder for a statement. The shared program captures the statements that are common to all the four versions ($\mathcal{O}$, $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{M}$), and the holes in $\hat{\mathcal{S}}$ represent program fragments that differ between the program versions. An edit $\Delta_{\mathcal{P}}$ for program version $\mathcal{P}$ represents a list of statements that will be substituted into the holes of the shared program to obtain $\mathcal{P}$.

Program version	$\mathcal{P}_v$	:=	$(\hat{\mathcal{S}}, \Delta)$
Edit	Δ	:=	$[\,] \mid \mathcal{S} :: \Delta$
Stmt with hole	$\hat{\mathcal{S}}$	:=	$[\cdot] \mid A \mid \hat{\mathcal{S}}_1; \hat{\mathcal{S}}_2 \mid C \,?\, \{\hat{\mathcal{S}}_1\} : \{\hat{\mathcal{S}}_2\}$
		|	$\text{while}(C)\ \{\hat{\mathcal{S}}\}$
Stmt	$\mathcal{S}$	:=	$A \mid \mathcal{S}_1; \mathcal{S}_2 \mid C \,?\, \{\mathcal{S}_1\} : \{\mathcal{S}_2\}$
		|	$\text{while}(C)\ \{\mathcal{S}\}$
Atom	A	:=	$\text{skip} \mid x := e \mid x[e_1] := e_2$

Fig. 5. Representation of program versions. Here, :: denotes list concatanation, and e and C represent expressions and predicates respectively.

Given this representation, we express *semantic conflict-freedom* as an assertion for each of the return variables (in this case, global variables modified by the `triggerActions` method). Since the `triggerActions` method modifies `time`, `value` and `queue`, we add an assertion for each of these variables. For instance, we add the following assertion on the value of `time` at exit from the four versions:

$$\begin{array}{c}(\texttt{time}_O = \texttt{time}_\mathcal{B} = \texttt{time}_\mathcal{A} = \texttt{time}_\mathcal{M}) \vee \\ \Big((\texttt{time}_O \neq \texttt{time}_\mathcal{A} \Rightarrow \texttt{time}_\mathcal{A} = \texttt{time}_\mathcal{M}) \wedge \\ (\texttt{time}_O \neq \texttt{time}_\mathcal{B} \Rightarrow \texttt{time}_\mathcal{B} = \texttt{time}_\mathcal{M})\Big)\end{array}$$

This assertion states that either (i) all four versions have identical side-effects on `time`, or (ii) if the side-effect on $\texttt{time}_\mathcal{A}$ (resp. $\texttt{time}_\mathcal{B}$) differs from $\texttt{time}_O$, then $\texttt{time}_\mathcal{M}$ in the merge should have identical side-effect as $\texttt{time}_\mathcal{A}$ (resp. $\texttt{time}_\mathcal{B}$). We add similar assertions for `value` and `queue`.

To prove these assertions, our method assumes that all four versions start out in identical states and then generates a *relational postcondition (RPC)* ψ such that the merge is semantically conflict-free if ψ logically implies the added assertions. Our RPC generation engine reasons about modifications over the base program by differentiating between three kinds of statements:

Shared statements. We summarize the behavior of shared statements using straight-line code snippets of the form $y = f(x_1, \ldots, x_n)$ where f is an uninterpreted function. Essentially, such a statement indicates that the value of variable y is some (unknown) function of variables $x_1, \ldots, x_n$. These "summaries" are generated using lightweight dependence analysis and allow our method to perform *abstract reasoning* over unchanged program fragments.

Holes. When our RPC generation engine encounters a hole in the shared program, it performs precise relational reasoning about different modifications by computing a *4-way product program* of the edits. As is well-known in the relational verification literature [Barthe et al. 2011, 2013], a product program $P_1 \times P_2$ is semantically equivalent to $P_1; P_2$ but is constructed in a way that facilitates the verification task. However, because product construction can result in a significant blow-up in program size, our technique generates *mini-products* in a novel way by considering each hole in isolation rather than constructing a full-fledged product of the four program versions.

Loops. Our RPC generation engine infers *relational loop invariants* for loops that contain edited program fragments. For instance, our method infers that (i) $\texttt{time}_O = \texttt{time}_\mathcal{B}$ and $\texttt{time}_\mathcal{A} = \texttt{time}_\mathcal{M}$, (ii) $\texttt{value}_O = \texttt{value}_\mathcal{A}$ and $\texttt{value}_\mathcal{B} = \texttt{value}_\mathcal{M}$, and (iii) the state of collection queue is identical in all four versions for the shared loop from Figure 4.

Using these ideas, our method is able to automatically generate an RPC that implies semantic conflict-freedom of this example. Furthermore, the entire procedure is push-button, including the generation of edits, RPC computation, and relational loop invariant generation.

3 REPRESENTATION OF PROGRAM VERSIONS

In this section, we describe our representation of program versions as *edits* applied to a *shared program with holes*. As shown in Figure 5, a program version $\mathcal{P}_v$ is a pair $(\hat{\mathcal{S}}, \Delta)$ where $\hat{\mathcal{S}}$ is a

$$\text{ApplyEdit}(\hat{\mathcal{S}}, \Delta) = \mathcal{S} \;\; \text{where}(\mathcal{S}, [\,]) = \text{Apply}(\hat{\mathcal{S}}, \Delta)$$

$$\text{Apply} :: (\hat{\mathcal{S}}, \Delta) \rightarrow (\mathcal{S}, \Delta')$$

$$\begin{array}{lcl}
\text{Apply}([\cdot], \mathcal{S} :: \Delta) & = & (\mathcal{S}, \Delta) \\
\text{Apply}(A, \Delta) & = & (A, \Delta) \\
\text{Apply}(\hat{\mathcal{S}}_1; \hat{\mathcal{S}}_2, \Delta) & = & \text{let } (\mathcal{S}_1, \Delta_1) = \text{Apply}(\hat{\mathcal{S}}_1, \Delta) \text{ in} \\
 & & \text{let } (\mathcal{S}_2, \Delta_2) = \text{Apply}(\hat{\mathcal{S}}_2, \Delta_1) \text{ in} \\
 & & ((\mathcal{S}_1; \mathcal{S}_2), \Delta_2) \\
\text{Apply}(C \;?\; \{\hat{\mathcal{S}}_1\} : \{\hat{\mathcal{S}}_2\}, \Delta) & = & \text{let } (\mathcal{S}_1, \Delta_1) = \text{Apply}(\hat{\mathcal{S}}_1, \Delta) \text{ in} \\
 & & \text{let } (\mathcal{S}_2, \Delta_2) = \text{Apply}(\hat{\mathcal{S}}_2, \Delta_1) \text{ in} \\
 & & (C \;?\; \{\mathcal{S}_1\} : \{\mathcal{S}_2\}, \Delta_2) \\
\text{Apply}(\text{while}(C)\; \{\hat{\mathcal{S}}\}, \Delta) & = & \text{let } (\mathcal{S}, \Delta') = \text{Apply}(\hat{\mathcal{S}}, \Delta) \text{ in} \\
 & & (\text{while}(C)\; \{\mathcal{S}\}, \Delta')
\end{array}$$

Fig. 6. Application of edit Δ to program with holes $\hat{\mathcal{S}}$.

statement with *holes* (i.e., missing statements) and an edit Δ is a list of statements (without holes). Given a program version $\mathcal{P}_v = (\hat{\mathcal{S}}, \Delta)$, we can obtain a full program $\mathcal{P} = \hat{\mathcal{S}}[\Delta]$ by applying the edit Δ to $\hat{\mathcal{S}}$ according to the ApplyEdit procedure of Figure 6. Effectively, ApplyEdit traverses the AST in depth-first order and replaces each hole with the next statement in the edit. Given n related programs $\mathcal{P}_1, \ldots, \mathcal{P}_n$, we assume the existence of a *diff* procedure that generates a shared program $\hat{\mathcal{S}}$ as well as n edits $\Delta_1, \ldots, \Delta_n$ such that $\forall i \in [1, n]$. $\text{ApplyEdit}(\hat{\mathcal{S}}, \Delta_i) = \mathcal{P}_i$. Since this *diff* procedure is orthogonal to our verification algorithm, we defer the discussion of our *diff* procedure until Section 6.

Since the language from Figure 5 uses standard imperative language constructs (including arrays), we assume an operational semantics described using judgments of the form $\sigma \vdash \mathcal{S} \Downarrow \sigma'$, where σ is a *valuation* that specifies the values of free variables in $\mathcal{S}$. Specifically, a valuation is a mapping from (variable, index) pairs to their corresponding values. The meaning of this judgment is that evaluating $\mathcal{S}$ under σ yields a new valuation σ'. In the rest of this paper, we also assume the existence of a special array called *out* that serves as the return value of the program. Any behavior that the programmer considers relevant (e.g., side effects or writing to the console) can be captured by storing the relevant values into this *out* array.

4 SEMANTIC CONFLICT FREEDOM

In this section, we first introduce *syntactic* conflict-freedom, which corresponds to the criterion used by many existing merge tools. We then explain why it falls short and formally describe the more robust notion of *semantic* conflict-freedom.

Definition 4.1. **(Syntactic conflict freedom)** Suppose that we are given four program versions $\mathcal{O} = (\hat{\mathcal{S}}, \Delta_{\mathcal{O}})$, $\mathcal{A} = (\hat{\mathcal{S}}, \Delta_{\mathcal{A}})$, $\mathcal{B} = (\hat{\mathcal{S}}, \Delta_{\mathcal{B}})$, $\mathcal{M} = (\hat{\mathcal{S}}, \Delta_{\mathcal{M}})$ representing the base program, the two variants, and the merge candidate respectively. We say that the merge candidate $\mathcal{M}$ is *syntactically conflict free* if the following conditions are satisfied for all $i \in [0, n)$, where n denotes the number of holes in $\hat{\mathcal{S}}$:

(1) If $\Delta_{\mathcal{O}}[i] \neq \Delta_{\mathcal{A}}[i]$, then $\Delta_{\mathcal{M}}[i] = \Delta_{\mathcal{A}}[i]$
(2) If $\Delta_{\mathcal{O}}[i] \neq \Delta_{\mathcal{B}}[i]$, then $\Delta_{\mathcal{M}}[i] = \Delta_{\mathcal{B}}[i]$
(3) Otherwise, $\Delta_{\mathcal{O}}[i] = \Delta_{\mathcal{A}}[i] = \Delta_{\mathcal{B}}[i] = \Delta_{\mathcal{M}}[i]$

Intuitively, the above definition states that the candidate merge $\mathcal{M}$ makes the same syntactic change as variant $\mathcal{A}$ (resp. $\mathcal{B}$) whenever $\mathcal{A}$ (resp. $\mathcal{B}$) differs from $\mathcal{O}$. While this definition may seem intuitively sensible, it does not accurately capture what it means for a merge candidate to be

correct. In particular, some incorrect merges may be conflict-free according to the above definition, while some correct merges may be rejected.

Example 4.2. Consider $\hat{\mathcal{S}} = [\cdot];[\cdot];out[0] := x$ and the edits $\Delta_O = [\text{skip}, \text{skip}]$, $\Delta_{\mathcal{A}} = [x := x + 1, \text{skip}]$, $\Delta_{\mathcal{B}} = [\text{skip}, x := x + 1]$, and $\mathcal{M} = [x := x + 1; x := x + 1]$. These programs are conflict-free according to the syntactic criterion given in Definition 4.1, but the merge is clearly incorrect (both variants increment x by 1, but the merge candidate ends up incrementing x by 2).

The above example illustrates that a syntactic notion of conflict freedom is not suitable for ruling out incorrect merges. Similarly, Definition 4.1 can also result in the rejection of perfectly valid merge candidates.

Example 4.3. Consider the base program $x > 0 \ ? \ \{y := 1\} : \{y := 0\}; \ out[0] := y$. Suppose this program has a bug that is caused by using the wrong predicate, so one variant fixes the bug by swapping the then and else branches, and the other variant changes the predicate from $x > 0$ to $x \leq 0$. Clearly, choosing either variant as the merge would be acceptable because they are semantically equivalent. However, assuming the shared program is $[\cdot]; out[0] := y$, there is no merge candidate that can satisfy Definition 4.1 because the two variants fill the hole in syntactically conflicting ways.

Based on the shortcomings of syntactic conflict freedom, we instead propose the following *semantic* variant:

Definition 4.4. **(Semantic conflict freedom)** Suppose that we are given four program versions $O, \mathcal{A}, \mathcal{B}, \mathcal{M}$ representing the base program, its two variants, and the merge candidate respectively. We say that $\mathcal{M}$ is *semantically conflict-free*, if for all valuations σ such that:

$$\sigma \vdash O \Downarrow \sigma_O \quad \sigma \vdash \mathcal{A} \Downarrow \sigma_{\mathcal{A}} \quad \sigma \vdash \mathcal{B} \Downarrow \sigma_{\mathcal{B}} \quad \sigma \vdash \mathcal{M} \Downarrow \sigma_{\mathcal{M}}$$

the following conditions hold for all i: [4]

(1) If $\sigma_O[(out, i)] \neq \sigma_{\mathcal{A}}[(out, i)]$, then $\sigma_{\mathcal{M}}[(out, i)] = \sigma_{\mathcal{A}}[(out, i)]$
(2) If $\sigma_O[(out, i)] \neq \sigma_{\mathcal{B}}[(out, i)]$, then $\sigma_{\mathcal{M}}[(out, i)] = \sigma_{\mathcal{B}}[(out, i)]$
(3) Otherwise, $\sigma_O[(out, i)] = \sigma_{\mathcal{A}}[(out, i)] = \sigma_{\mathcal{B}}[(out, i)] = \sigma_{\mathcal{M}}[(out, i)]$

In contrast to syntactic conflict freedom, Definition 4.4 requires agreement between the *values* that are returned by the program. Specifically, it says that, if the i'th value returned by variant A (resp. B) differs from the i'th value returned by base, then the i'th return value of the merge should agree with A (resp. B). According to this definition, the merge candidate from Example 4.2 is *not* conflict-free because it returns 2 whereas both variants return 1. Furthermore, for Example 4.3, we can find a merge candidate (e.g., one of the variants) that satisfies semantic conflict freedom.

Discussion. Our definition of semantic conflict freedom is intended as a *sufficient* –rather than necessary– condition for correctness. In particular, there may be situations where developers consider a 3-way merge to be correct even though it does not satisfy Definition 4.4. However, since it is not possible to make this judgement without additional input from the programmer (e.g., the invariant that must be respected by the candidate merge), we believe that developers should be made aware of any violation of Definition 4.4.

5 VERIFYING SEMANTIC CONFLICT FREEDOM

We now turn our attention to the verification algorithm for proving semantic conflict-freedom. The high-level structure of this algorithm is shown in Algorithm 1. The procedure VERIFY takes as input

[4]We assume that $out[i]$ is a special value $\perp$ if $(out, i) \notin dom(\sigma)$

Algorithm 1 Algorithm for verifying conflict freedom

1: **procedure** VERIFY($\hat{\mathcal{S}}, \Delta_1, \Delta_2, \Delta_3, \Delta_4$)
2: **assume** $vars(\{\hat{\mathcal{S}}[\Delta_1], \ldots, \hat{\mathcal{S}}[\Delta_4]\}) = V$
3: $\varphi := (V_1 = V_2 \wedge V_1 = V_3 \wedge V_1 = V_4)$
4: $\psi :=$ RelationalPost($\hat{\mathcal{S}}, \Delta_1, \Delta_2, \Delta_3, \Delta_4, \varphi$)
5: $\chi_1 := \forall i.\ (out_1[i] \neq out_2[i] \Rightarrow out_2[i] = out_4[i])$
6: $\chi_2 := \forall i.\ (out_1[i] \neq out_3[i] \Rightarrow out_3[i] = out_4[i])$
7: $\chi_3 := \forall i.\ (out_1[i] = out_2[i] = out_3[i] = out_4[i])$
8: **return** Valid($\psi \Rightarrow (\chi_1 \wedge \chi_2) \vee \chi_3$)

a shared program (with holes) $\hat{\mathcal{S}}$, an edit Δ_1 for the base program, edits Δ_2, Δ_3 for the variants, and an edit Δ_4 for the merge candidate. Conceptually, the algorithm consists of three steps:

Precondition. Algorithm 1 starts by generating a pre-condition φ (line 3) stating that all variables initially have the same value. [5] Here, V_1 denotes the variables in the base program, V_2, V_3 denote variables in the variants, and V_4 refers to variables in the merge candidate. We use the notation $V_i = V_j$ as short-hand for $\forall v \in V.\ v_i = v_j$.

RPC computation. The next step of the algorithm is to compute a *relational post-condition* ψ of φ with respect to the four program versions (line 4). Such a relational post-condition ψ states relationships between variables V_1, V_2, V_3, and V_4 and has the property that it is also post-condition of the program $(\hat{\mathcal{S}}[\Delta_1])[V_1/V]; \ldots; (\hat{\mathcal{S}}[\Delta_4])[V_4/V]$. We will explain the RelationalPost procedure in detail shortly.

Checking conflict freedom. The last step of the algorithm checks whether the relational post-condition ψ logically implies semantic conflict freedom (line 8). Specifically, observe that the constraint $(\chi_1 \wedge \chi_2) \vee \chi_3$ encodes precisely the three conditions from Definition 4.4, so the program is conflict-free if ψ implies $(\chi_1 \wedge \chi_2) \vee \chi_3$.

5.1 Computing Relational Postconditions

Since the core part of the verification algorithm is the computation of RPCs, we now describe the RelationalPost procedure. As mentioned in Section 1, the key idea is to analyze edits in a precise way by constructing product programs, but perform lightweight reasoning for shared program parts using dependence analysis.

Our RPC generation engine is described in Figure 7 using judgments $\vec{\Delta}, \varphi \vdash \hat{\mathcal{S}} : \varphi', \vec{\Delta}'$. Here, φ is a precondition relating variables in different program versions, and $\vec{\Delta}$ is a vector of n edits applied to a shared base program $\hat{\mathcal{S}}$. The meaning of this judgment is that the following Hoare triple is valid:

$$\{\varphi\}\ \hat{\mathcal{S}}[\Delta_1][V_1/V]; \ldots; \hat{\mathcal{S}}[\Delta_n][V_n/V]\ \{\varphi'\}$$

In other words, φ' is a sound relational post-condition of the four program versions with respect to precondition φ. Since the edits in $\vec{\Delta}$ may contain more statements than there are holes in $\hat{\mathcal{S}}$, we use $\vec{\Delta}'$ to denote the remaining edits that were not "used" while analyzing $\hat{\mathcal{S}}$.

Let us now consider the rules in Figure 7 in more detail. The first rule corresponds to the case where we encounter a hole in the shared program and need to analyze the edits. In this case, we construct a "mini" product program $\mathcal{S}$ that describes the simultaneous execution of the edits. As

[5] Observe that this precondition also applies to local variables, not just arguments, and allows our technique to handle cases in which one of the variants introduces a new variable.

$$(1)\quad \frac{\mathcal{S} = \mathsf{head}(\Delta_1)[V_1/V] \circledast \ldots \circledast \mathsf{head}(\Delta_4)[V_4/V]}{\vec{\Delta}, \varphi \vdash [\cdot] : \mathit{post}(\mathcal{S}, \varphi), [\mathsf{tail}(\Delta_1), \ldots, \mathsf{tail}(\Delta_4)]}$$

$$(2)\quad \frac{\begin{array}{c}\mathsf{Modifies}(\mathcal{S}) = \{y_1, \ldots, y_n\} \\ \vec{x_i} = \mathsf{Dependencies}(\mathcal{S}, y_i) \\ \mathcal{S}_i = (y_i := F_i(\vec{x_i}))[V_1/V]; \ldots; (y_i := F_i(\vec{x_i}))[V_4/V]\end{array}}{\vec{\Delta}, \varphi \vdash \mathcal{S} : \mathit{post}(\mathcal{S}_1; \ldots; \mathcal{S}_n, \varphi), \vec{\Delta}}$$

$$(3)\quad \frac{\vec{\Delta}, \varphi \vdash \hat{\mathcal{S}}_1 : \varphi', \vec{\Delta}' \quad \vec{\Delta}', \varphi' \vdash \hat{\mathcal{S}}_2 : \varphi'', \vec{\Delta}''}{\vec{\Delta}, \varphi \vdash \hat{\mathcal{S}}_1; \hat{\mathcal{S}}_2 : \varphi'', \vec{\Delta}''}$$

$$(4)\quad \frac{\begin{array}{c}\varphi \models \bigwedge_{i,j} C[V_i/V] \leftrightarrow C[V_j/V] \\ \vec{\Delta}, \varphi \wedge C[V_1/V] \vdash \hat{\mathcal{S}}_1 : \varphi', \vec{\Delta}' \\ \vec{\Delta}', \varphi \wedge \neg C[V_1/V] \vdash \hat{\mathcal{S}}_2 : \varphi'', \vec{\Delta}''\end{array}}{\vec{\Delta}, \varphi \vdash C \,?\, \{\hat{\mathcal{S}}_1\} : \{\hat{\mathcal{S}}_2\} : \varphi' \vee \varphi'', \vec{\Delta}''}$$

$$(5)\quad \frac{\begin{array}{c}\varphi \models \mathcal{I} \quad \vec{\Delta}, \mathcal{I} \wedge \bigwedge_i C[V_i/V] \vdash \hat{\mathcal{S}} : \mathcal{I}', \vec{\Delta}' \quad \mathcal{I}' \models \mathcal{I} \\ \mathcal{I} \models \bigwedge_{i,j} C[V_i/V] \leftrightarrow C[V_j/V]\end{array}}{\vec{\Delta}, \varphi \vdash \mathsf{while}(C)\ \ \hat{\mathcal{S}} : \mathcal{I} \wedge \bigwedge_i \neg C[V_i/V], \vec{\Delta}'}$$

$$(6)\quad \frac{\begin{array}{c}\mathcal{S} = (\hat{\mathcal{S}}[\Delta_1])[V_1/V] \circledast \ldots \circledast (\hat{\mathcal{S}}[\Delta_4])[V_4/V] \\ \Delta_i = (\Delta_i^1 :: \Delta_i^2)\ \ (|\Delta_i^1| = \mathsf{numHoles}(\hat{\mathcal{S}}))\end{array}}{\vec{\Delta}, \varphi \vdash \hat{\mathcal{S}} : \mathit{post}(\mathcal{S}, \varphi), [\Delta_1^2, \ldots, \Delta_4^2]}$$

Fig. 7. RPC inference.

we will see in Section 5.2, an n-way product program $\mathcal{S}_1 \circledast \ldots \circledast \mathcal{S}_n$ is semantically equivalent to the sequential composition $\mathcal{S}_1; \ldots; \mathcal{S}_n$ but has the advantage of being easier to analyze. Given such a "mini product" $\mathcal{S}$, our RPC generation engine computes the post-condition of $\mathcal{S}$ in the standard way using a *post* function, where $\mathit{post}(\mathcal{S}, \varphi)$ yields a sound post-condition of φ with respect to $\mathcal{S}$. Since $\mathcal{S}$ may contain loops in the general case, the computation of *post* may require loop invariant generation. As we discuss in Section 5.2, the key advantage of constructing a product program is to facilitate loop invariant generation using standard techniques.

Rule (2) corresponds to the case where we encounter a program fragment $\mathcal{S}$ without holes. Since $\mathcal{S}$ has not been modified by any of the variants, we analyze $\mathcal{S}$ in a lightweight way using dependence analysis. Specifically, for each variable y_i that is modified by $\mathcal{S}$, we compute the set of variables $x_1, \ldots, x_k$ that it depends on. We then summarize the behavior of $\mathcal{S}$ using statements of the form $y_i = F_i(x_1, \ldots, x_k)$ where F_i is a fresh uninterpreted function symbol. Hence, rather than analyzing the entire code fragment $\mathcal{S}$ (which could potentially be very large), we analyze its behavior in a lightweight way by modeling it as straight-line code over uninterpreted functions. [6]

Rule (3) for sequencing is similar to its corresponding proof rule in standard Hoare logic: Given a statement $\hat{\mathcal{S}}_1; \hat{\mathcal{S}}_2$, we first compute the relational post-condition φ' of $\hat{\mathcal{S}}_1$ and then use φ' as the precondition for $\hat{\mathcal{S}}_2$. Since $\hat{\mathcal{S}}_1$ and $\hat{\mathcal{S}}_2$ may contain edits nested inside them, this proof rule combines reasoning about $\hat{\mathcal{S}}_1$ and $\hat{\mathcal{S}}_2$ in a precise, yet lightweight way, without constructing a 4-way product for the entire program.

[6] There are rare cases in which this abstraction would lead to imprecision. Section 7 describes how our implementation handles such cases.

Rule (4) allows us to analyze conditionals $C\ ?\ \{\hat{\mathcal{S}}_1\} : \{\hat{\mathcal{S}}_2\}$ in a modular way whenever possible. As in the sequencing case, we would like to analyze $\hat{\mathcal{S}}_1$ and $\hat{\mathcal{S}}_2$ in isolation and then combine the results. Unfortunately, such compositional reasoning is only possible if all program versions take the same path. For instance, consider the shared program $[\cdot]; x > 0\ ?\ \{y := 1\} : \{y := 2\}$ and two versions A, B given by the edits $[x := y]$ and $[x := z]$. Since A could take the then branch while B takes the else branch (or vice versa), we need to reason about all possible combinations of paths. Hence, the first premise of this rule checks whether each $C[V_i/V]$ can be proven to be equivalent to all other $C[V_j/V]$'s under precondition φ. If this is the case, all program versions take the same path, so we can reason compositionally. Otherwise, our analysis falls back upon the conservative, but non-modular, proof rule (6) that we will explain shortly.

Rule (5) uses *inductive relational invariants* for loops that have been edited in different ways by each program variant. Specifically, the first premise of this rule states that the relational invariant $\mathcal{I}$ is implied by the loop pre-condition, and the next two premises enforce that $\mathcal{I}$ is preserved by the loop body (i.e., $\mathcal{I}$ is inductive). Thus, assuming that all loops execute the same number of times (checked by line 2 of rule 5), we can conclude that $\mathcal{I} \wedge \bigwedge_i \neg C[V_i/V]$ holds after the loop. Note that rule (5) does not describe how to compute such relational loop invariants; it simply asserts that $\mathcal{I}$ is inductive. As will be described in Section 7, our implementation uses predicate abstraction to infer such relational loop invariants.

Rule (6) allows us to fall back upon non-modular reasoning when it is not sound to analyze edits in a compositional way. Given a statement $\hat{\mathcal{S}}$ with holes, rule (6) constructs the product program $(\hat{\mathcal{S}}[\Delta_1])[V_1/V] \circledast \ldots \circledast (\hat{\mathcal{S}}[\Delta_4])[V_4/V]$ and computes its post-condition in the standard way. While rule (6) is a generalization of rule (1), it is *only* used in cases where compositional reasoning is unsound. In particular, since product construction can cause *significant* blow up in program size, the use of modular reasoning is very important for the practicality of our approach (see Section 8).

Theorem 5.1. **(Soundness of relational post-condition)** [7] *Let $\hat{\mathcal{S}}$ be a shared program with holes and $\vec{\Delta}$ be the edits such that $|\Delta_i| = \mathsf{numHoles}(\hat{\mathcal{S}})$. Let φ' be the result of calling $\mathsf{RelationalPost}(\hat{\mathcal{S}}, \vec{\Delta}, \varphi)$ (i.e., $\vec{\Delta}, \varphi \vdash \hat{\mathcal{S}} : \varphi', []$ according to Figure 7). Then, the following Hoare triple is valid:*

$$\{\varphi\}\ (\hat{\mathcal{S}}[\Delta_1])[V_1/V]; \ldots; (\hat{\mathcal{S}}[\Delta_n])[V_n/V]\ \{\varphi'\}$$

5.2 Construction of Product Programs

In this section, we describe our method for constructing n-way product programs. While there are several strategies for generating 2-way product programs in the literature (e.g., [Barthe et al. 2011, 2013]), our method generalizes these techniques to n-way products and uses a program similarity metric to guide product construction. As a result, our method can generate verification-friendly product programs while obviating the need to perform backtracking search over non-deterministic product construction rules.

Before we describe our product construction technique, we first give a simple example to illustrate how product construction facilitates relational verification:

Example 5.2. Consider the following programs $\mathcal{S}_1$ and $\mathcal{S}_2$:

$$\begin{array}{ll} \mathcal{S}_1: & i_1 := 0;\ \mathsf{while}(i_1 < n_1)\ \{i_1 := i_1 * x_1\} \\ \mathcal{S}_2: & i_2 := 0;\ \mathsf{while}(i_2 < n_2)\ \{i_2 := i_2 * x_2\} \end{array}$$

and the precondition $n_1 = n_2 \wedge x_1 = x_2$. It is easy to see that i_1 and i_2 will have the same value after executing $\mathcal{S}_1$ and $\mathcal{S}_2$. Now, consider analyzing the program $\mathcal{S}_1; \mathcal{S}_2$. While a static analyzer can *in principle* infer this post-condition by coming up with a precise loop invariant that captures the

[7] Proofs of all theorems are available in the Appendix.

exact symbolic value of i_1 and i_2 during each iteration, this is clearly a very difficult task. To see why product programs are useful, now consider the following program $\mathcal{S}$:

$$
\begin{array}{ll}
(1) & i_1 := 0; i_2 := 0; \\
(2) & \text{while}(i_1 < n_1 \wedge i_2 < n_2)\ \{i_1 := i_1 * x_1;\ i_2 := i_2 * x_2;\} \\
(3) & (i_1 < n_1)?\{\text{while}(i_1 < n_1)\ \{i_1 := i_1 * x_1\}\} \\
 & \qquad : \{(i_2 < n_2)?\{\text{while}(i_2 < n_2)\ \{i_2 := i_2 * x_2\}\} : \{\text{skip}\}\}
\end{array}
$$

Here, $\mathcal{S}$ is equivalent to $\mathcal{S}_1; \mathcal{S}_2$ because it executes both loops in lockstep until one of them terminates and then executes the remainder of the other loop. While this code may look complicated, it is much easier to statically reason about $\mathcal{S}$ than $\mathcal{S}_1; \mathcal{S}_2$. In particular, since $i_1 = i_2 \wedge x_1 = x_2 \wedge n_1 = n_2$ is an inductive invariant of the first loop in $\mathcal{S}$, we can easily prove that line (3) is dead code and that $i_1 = i_2$ is a valid post-condition of $\mathcal{S}$. As this example illustrates, product programs can make relational verification easier by executing loops from different programs in lockstep.

Our n-way product construction method is presented in Figure 8 using inference rules that derive judgments of the form $\vdash \mathcal{S}_1 \circledast \ldots \circledast \mathcal{S}_n \rightsquigarrow \mathcal{S}$ where programs $\mathcal{S}_1, \ldots, \mathcal{S}_n$ *do not share any variables* (i.e., each $\mathcal{S}_i$ refers to variables V_i such that $V_i \cap V_j = \emptyset$ for $i \neq j$). The generated product $\mathcal{S}$ is semantically equivalent to $\mathcal{S}_1; \ldots; \mathcal{S}_n$ but is constructed in a way that makes $\mathcal{S}$ easier to be statically analyzed. Similar to prior relational verification techniques, the key idea is to synchronize loops from different program versions as much as possible. However, our method differs from existing techniques in that it combines different snippets of arbitrarily many programs and uses a program similarity metric to guide this construction.

guide product construction and generalizes them to n-way products.

Notation. Before discussing the product construction algorithm summarized in Figure 8, we first introduce some useful notation: We abbreviate $\mathcal{S}_1 \circledast \ldots \circledast \mathcal{S}_n$ using the notation $\mathsf{P}^{\circledast}$, and we write P to denote the list $(\mathcal{S}_1, \ldots, \mathcal{S}_n)$. Also, given a statement $\mathcal{S}$, we write $\mathcal{S}[i]$ to denote the i'th element in the sequence (i.e., $\mathcal{S}[0]$ denotes the first element).

Product construction algorithm. We are now ready to explain the product construction rules shown in Figure 8. Rule (1) is quite simple and deals with the case where the first program starts with an atomic statement A. Since we can always compute a precise post-condition for atomic statements, it is not necessary to "synchronize" A with any of the statements from other programs. Therefore, we first compute the product program $\mathcal{S}_1 \circledast \mathsf{P}^{\circledast}$, i.e. $\mathcal{S}_1 \circledast \mathcal{S}_2 \circledast \ldots \circledast \mathcal{S}_n$, and then sequentially compose it with A.

Rule (2) considers the case where the first program starts with a conditional $C\ ?\ \{\mathcal{S}_t\} : \{\mathcal{S}_e\}$. In general, $\mathcal{S}_t$ and $\mathcal{S}_e$ may contain loops; so, there may be an opportunity to synchronize any loops within $\mathcal{S}_t$ and $\mathcal{S}_e$ with loops from $\mathsf{P} = \mathcal{S}_2, \ldots, \mathcal{S}_n$. Therefore, we construct the product program as $C\ ?\ \{\mathcal{S}'\} : \{\mathcal{S}''\}$ where $\mathcal{S}'$ (resp. $\mathcal{S}''$) is the product of the then (resp. else) branch with $\mathsf{P}^{\circledast}$. Observe that this rule can cause a blow-up in program size because we embed the continuation program $\mathcal{S}_1$ inside both the then *and* else branches of the conditional. However, our overall verification algorithm tries to minimize this blow-up by *only* constructing product programs for edited program fragments or in cases where compositional reasoning would otherwise be unsound.

All of the remaining rules in Figure 8 deal with loops, with the goal of simplifying invariant generation. Specifically, rule (3) considers the case where the first program starts with a loop but there is some program $\mathcal{S}_i$ in $\mathsf{P} = (\mathcal{S}_2, \ldots, \mathcal{S}_n)$ that does not start with a loop. In this case, we want to "get rid of" program $\mathcal{S}_i$ by using rules (1) and (2); thus, we move $\mathcal{S}_i$ to the beginning and construct the product program $\mathcal{S}$ for $\mathcal{S}_i \circledast (\mathsf{P} \backslash \mathcal{S}_i)^{\circledast} \circledast \text{while}(C_1)\ \{\mathcal{S}_{B_1}\}; \mathcal{S}_1$.

$$(1)\quad \frac{\vdash \mathcal{S}_1 \circledast \mathsf{P}^{\circledast} \rightsquigarrow \mathcal{S}}{\vdash A; \mathcal{S}_1 \circledast \mathsf{P}^{\circledast} \rightsquigarrow A; \mathcal{S}}$$

$$(2)\quad \frac{\vdash \mathcal{S}_t; \mathcal{S}_1 \circledast \mathsf{P}^{\circledast} \rightsquigarrow \mathcal{S}' \quad \vdash \mathcal{S}_e; \mathcal{S}_1 \circledast \mathsf{P}^{\circledast} \rightsquigarrow \mathcal{S}''}{\vdash (C\ ?\ \{\mathcal{S}_t\} : \{\mathcal{S}_e\}); \mathcal{S}_1 \circledast \mathsf{P}^{\circledast} \rightsquigarrow (C\ ?\ \{\mathcal{S}'\} : \{\mathcal{S}''\})}$$

$$(3)\quad \frac{\begin{array}{c}\exists \mathcal{S}_i \in \mathsf{P}.\ \mathcal{S}_i[0] \neq \text{while}(C_i)\ \{\mathcal{S}_{B_i}\} \\ \vdash \mathcal{S}_i \circledast (\mathsf{P} \setminus \mathcal{S}_i)^{\circledast} \circledast (\text{while}(C_1)\ \{\mathcal{S}_{B_1}\}); \mathcal{S}_1 \rightsquigarrow \mathcal{S}\end{array}}{\vdash (\text{while}(C_1)\ \{\mathcal{S}_{B_1}\}); \mathcal{S}_1 \circledast \mathsf{P}^{\circledast} \rightsquigarrow \mathcal{S}}$$

$$(4)\quad \frac{\begin{array}{c}\forall \mathcal{S}_i \in \mathsf{P}.\ \mathcal{S}_i[0] = \text{while}(C_i)\ \{\mathcal{S}_{B_i}\} \\ \exists \mathsf{H} \subseteq \mathsf{P}.\ \forall \mathsf{L} \subseteq \mathsf{P}.\ sim(\mathsf{H}) \geq sim(\mathsf{L}) \\ \vdash (\mathsf{H}[0])^{\circledast} \rightsquigarrow \mathcal{S}' \quad \vdash (\mathsf{H}[1\ldots])^{\circledast} \circledast (\mathsf{P} \setminus \mathsf{H})^{\circledast} \rightsquigarrow \mathcal{S}''\end{array}}{\mathsf{P}^{\circledast} \rightsquigarrow \mathcal{S}'; \mathcal{S}''}$$

$$(5)\quad \frac{\begin{array}{c}\vdash \mathcal{S}_{B_1} \circledast \mathcal{S}_{B_2} \rightsquigarrow \mathcal{S} \\ W := \text{while}(C_1 \wedge C_2)\ \{\mathcal{S}\} \\ R := C_1\ ?\ \{\text{while}(C_1)\ \{\mathcal{S}_{B_1}\}\} : \{(C_2\ ?\ \{\text{while}(C_2)\ \{\mathcal{S}_{B_2}\}\} : \{\text{skip}\})\} \\ \vdash W; R \circledast \mathsf{P}^{\circledast} \rightsquigarrow \mathcal{S}'\end{array}}{\vdash (\text{while}(C_1)\ \{\mathcal{S}_{B_1}\}) \circledast (\text{while}(C_2)\ \{\mathcal{S}_{B_2}\}) \circledast \mathsf{P}^{\circledast} \rightsquigarrow \mathcal{S}'}$$

Fig. 8. Product construction. The base case is the trivial rule $\vdash \mathcal{S} \rightsquigarrow \mathcal{S}$, and we assume that every program ends in a *skip* and that *skip* $\circledast\ \mathsf{P}^{\circledast}$ is the same as $\mathsf{P}^{\circledast}$.

Before we continue to the other rules, we make two important observations about rule (3). First, this rule exploits the commutativity and associativity of the $\circledast$ operator [8]; however, it uses these properties in a restricted form by applying them only where they are useful. Second, after exhaustively applying rules (1), (2), and (3) on some $\mathsf{P}_0^{\circledast}$, note that we will end up with a new $\mathsf{P}_1^{\circledast}$ where *all* programs in P_1 are guaranteed to start with a loop.

Rule (4) considers the case where all programs start with a loop and utilizes a similarity metric *sim* to identify which loops to synchronize. [9] In particular, let H be the subset of the programs in P that are "most similar" according to our similarity metric. Since all programs in H start with a loop, we first construct the product program $\mathcal{S}'$ of these loops. We then construct the product program $\mathcal{S}''$ for the remaining programs $\mathsf{P} \setminus H$ and the remaining parts of the programs in H.

The final rule (5) defines what it means to "execute loops in lockstep as much as possible". Given two programs that start with loops $\text{while}(C_1)\ \{\mathcal{S}_1\}$ and $\text{while}(C_2)\ \{\mathcal{S}_2\}$, we first construct the product $\mathcal{S}_1 \circledast \mathcal{S}_2$ and generate the synchronized loop as $\text{while}(C_1 \wedge C_2)\ \{\mathcal{S}_1 \circledast \mathcal{S}_2\}$. Since these loops may not execute the same number of times, we still need to generate the "continuation" R, which executes any remaining iterations of one of the loops. Thus, $W; R$ in rule (5) is semantically equivalent to $\text{while}(C_1)\ \{\mathcal{S}_1\}; \text{while}(C_2)\ \{\mathcal{S}_2\}$. Now, since there may be further synchronization opportunities between $W; R$ and the remaining programs $\mathcal{S}_3, \ldots, \mathcal{S}_n$, we obtain the final product program by computing $W; R \circledast \mathcal{S}_3 \circledast \ldots \circledast \mathcal{S}_n$.

Example 5.3. Consider again the programs $\mathcal{S}_1$ and $\mathcal{S}_2$ from Example 5.2. We can use rules (1) and (5) from Figure 8 to compute the product program for $\mathcal{S}_1 \circledast \mathcal{S}_2$. The resulting product is exactly the program $\mathcal{S}$ shown in Example 5.2.

[8] Recall that different programs do not share variables

[9] While our product construction rules can work with any similarity metric, our implementation uses edit distance to implement *sim*.

Since rules (4) or (5) are both applicable when all programs start with a loop, our product construction algorithm first applies rule (4) and then uses rule (5) when constructing the product for $(H[0])^{\circledast}$ in rule (4). Thus, our method ensures that loops that are most similar to each other are executed in lockstep, which in turn greatly facilitates verification.

Theorem 5.4. **(Soundness of product)** *Let $\mathcal{S}_1, \ldots, \mathcal{S}_n$ be statements with disjoint variables, and let $\vdash \mathcal{S}_1 \circledast \ldots \circledast \mathcal{S}_n \rightsquigarrow \mathcal{S}$ according to Figure 8. Then, for all valuations σ, we have $\sigma \vdash \mathcal{S}_1; \ldots; \mathcal{S}_n \Downarrow \sigma'$ iff $\sigma \vdash \mathcal{S} \Downarrow \sigma'$.*

6 EDIT GENERATION

The verification algorithm we described in Section 5 requires all program versions to be represented as edits applied to a shared program with holes. This representation is very important because it allows our verification algorithm to reason about modifications to different program parts in a compositional way. In this section, we describe an n-way AST differencing algorithm that can be used to generate the desired program representation.

Our n-way diff algorithm is presented in Algorithm 2. Procedure NDiff takes as input n programs $\mathcal{S}_1, \ldots, \mathcal{S}_n$ and returns a pair $(\hat{\mathcal{S}}, \vec{\Delta})$ where $\hat{\mathcal{S}}$ is a shared program with holes and $\vec{\Delta}$ is a list of edits such that $\hat{\mathcal{S}}[\Delta_i] = \mathcal{S}_i$. The loop inside the NDiff procedure maintains the key invariant $\forall j.\ 1 \leq j < i \Rightarrow \hat{\mathcal{S}}[\Delta_j] = \mathcal{S}_j$. Thus, upon termination, NDiff guarantees that $\hat{\mathcal{S}}[\Delta_i] = \mathcal{S}_i$ for all $i \in [1, n]$.

The bulk of the work of the NDiff procedure is performed by the auxiliary GenEdit function, which uses a 2-way AST differencing algorithm to extend the diff from k to $k + 1$ programs. Specifically, GenEdit takes as input a new program $\mathcal{S}$ as well as the diff of the first k programs, where the diff is represented as a shared program $\hat{\mathcal{S}}$ with holes as well as edits $\Delta_1, \ldots, \Delta_k$. The key idea underlying GenEdit is to use a standard 2-way AST diff algorithm to compute the diff between $\hat{\mathcal{S}}$ and the new program $\mathcal{S}$ and then use the result to update the existing edits $\Delta_1, \ldots, \Delta_k$.

In more detail, the Diff2 procedure used in GenEdit yields the 2-way diff of $\hat{\mathcal{S}}$ and $\mathcal{S}$ as a triple $(\hat{\mathcal{S}}', \Delta, \hat{\Delta})$ such that $\hat{\mathcal{S}}'[\Delta] = \mathcal{S}$ and $\hat{\mathcal{S}}'[\hat{\Delta}] = \hat{\mathcal{S}}$. [10] The core insight underlying GenEdit is to use $\hat{\Delta}$ to update the existing edits $\Delta_1, \ldots, \Delta_k$ for the first k programs. Specifically, we use a procedure Compose to combine each existing edit Δ_i with the output $\hat{\Delta}$ of Diff2. The Compose procedure is defined recursively and inspects the first element of $\hat{\Delta}$ in each recursive call. If the first element is a hole, we preserve the existing edit; otherwise, we use the edit from $\hat{\Delta}$. Thus, if $\text{Compose}(\hat{\Delta}, \Delta_i)$ yields Δ_i', we have $\hat{\mathcal{S}}'[\Delta_i'] = \hat{\mathcal{S}}[\Delta_i]$. In other words, the Compose procedure allows us to update the diff of the first k programs to generate a sound diff of $k + 1$ programs.

Theorem 6.1. **(Soundness of NDiff)** *Let $\text{NDiff}(\mathcal{S}_1, \ldots, \mathcal{S}_n)$ be $(\hat{\mathcal{S}}, \vec{\Delta})$. Then we have $\hat{\mathcal{S}}[\Delta_i] = \mathcal{S}_i$ for all $i \in [1, n]$.*

7 IMPLEMENTATION

We implemented the techniques proposed in this paper in a tool called SafeMerge for checking semantic conflict-freedom of Java programs. SafeMerge is written in Haskell and uses the Z3 SMT solver [De Moura and Bjørner 2008]. In what follows, we describe relational invariant generation, our handling of various aspects of the Java language and other implementation choices.

Relational invariant generation. The RPC computation engine from Section 5.1 requires an inductive loop invariant relating variables from the four program versions. Our implementation

[10] Existing 2-way AST diff algorithms can be adapted to produce diffs in this form. We provide our Diff2 implementation under supplementary materials.

Algorithm 2 n-way AST differencing algorithm

1: **procedure** NDiff($\mathcal{S}_1, \ldots, \mathcal{S}_n$)
2: $\hat{\mathcal{S}} \leftarrow \mathcal{S}_1$; $\vec{\Delta} \leftarrow []$; $i \leftarrow 2$;
3: **while** $i \leq n$ **do**
4: $(\hat{\mathcal{S}}, \vec{\Delta}) \leftarrow \text{GenEdit}(\hat{\mathcal{S}}, \mathcal{S}_i, \vec{\Delta})$
5: **return** $(\hat{\mathcal{S}}, \vec{\Delta})$
6: **procedure** GenEdit($\hat{\mathcal{S}}, \mathcal{S}, \Delta_1, \ldots, \Delta_k$)
7: **Requires:** $|\Delta_i| = \text{numHoles}(\hat{\mathcal{S}})$ for all $i \in [1, \ldots, k]$
8: **Ensures:** $|\Delta'_i| = \text{numHoles}(\hat{\mathcal{S}}')$ for $i \in [1, \ldots, k+1]$
9: **Ensures:** $\hat{\mathcal{S}}'[\Delta'_{k+1}] = \mathcal{S}$ and $\hat{\mathcal{S}}'[\Delta'_i] = \hat{\mathcal{S}}[\Delta_i]$ for $i \in [1, \ldots, k]$
10: $(\hat{\mathcal{S}}', \Delta, \hat{\Delta}) := \text{Diff2}(\mathcal{S}, \hat{\mathcal{S}})$
11: **for** i in $[1, k]$ **do**
12: $\Delta'_i := \text{Compose}(\hat{\Delta}, \Delta_i)$
13: **return** $(\hat{\mathcal{S}}', \Delta'_1, \ldots, \Delta'_k, \Delta)$
14: **procedure** Compose($\hat{\Delta}, \Delta$)
15: **Requires:** $|\Delta| = \text{numHoles}(\hat{\Delta})$
16: **Output:** Edit Δ'
17: **Ensures:** $|\Delta'| = |\hat{\Delta}|$
18: **Ensures:** For any $\hat{\mathcal{S}}$ s.t. $\text{numHoles}(\hat{\mathcal{S}}) = |\hat{\Delta}|$, $(\hat{\mathcal{S}}[\hat{\Delta}])[\Delta] = \hat{\mathcal{S}}[\Delta']$
19: **if** $\hat{\Delta} = [\,]$ **then return** $[\,]$
20: **else if** $\text{head}(\hat{\Delta}) = [\cdot]$ **then**
21: **return** $\text{head}(\Delta) :: \text{Compose}(\text{tail}(\hat{\Delta}), \text{tail}(\Delta))$
22: **else return** $\text{head}(\hat{\Delta}) :: \text{Compose}(\text{tail}(\hat{\Delta}), \Delta)$
23: **procedure** Diff2($\mathcal{S}, \hat{\mathcal{S}}$)
24: **Input:** A program $\mathcal{S}$ and a shared program $\hat{\mathcal{S}}$
25: **Output:** Shared program $\hat{\mathcal{S}}'$ and edits $\Delta, \hat{\Delta}$
26: **Ensures:** $|\Delta| = |\hat{\Delta}| = \text{numHoles}(\hat{\mathcal{S}}')$
27: **Ensures:** $\text{numHoles}(\hat{\Delta}) = \text{numHoles}(\hat{\mathcal{S}})$
28: **Ensures:** $\hat{\mathcal{S}}'[\Delta] = \mathcal{S}, \hat{\mathcal{S}}'[\hat{\Delta}] = \hat{\mathcal{S}}$

automatically infers relational loop invariants using the Houdini framework for (monomial) predicate abstraction [Flanagan and Leino 2001]. Specifically, we consider predicate templates of the form $x_i = x_j$ relating values of the same variable from different program versions, and compute the strongest conjunct that satisfies the conditions of rule (5) of Figure 7.

Modeling the heap and collections. As standard in prior verification literature [Flanagan et al. 2002], we model each field f in the program as follows: We introduce a map f from object identifiers to values and model reads and writes to the map using the *select* and *update* functions in the theory of arrays. Similarly, our implementation models collections, such as `ArrayList` and `Queue`, using arrays. Specifically, we use an array to represent the contents of the collection and use scalar variables to model the size of the collection as well as the current position of an iterator over the collection [Dillig et al. 2011].

Side effects of a method. Our formalization uses an `out` array to model all relevant side effects of a method. Since real Java programs do not contain such a construct, our implementation checks

semantic conflict freedom on the method's return value, the final state of the receiver object as well as any field modified in the method.

Analysis of shared statements. Recall that our technique abstracts away shared program statements using uninterpreted functions (rule (2) from Figure 7). However, because unconditional use of such abstraction can result in false positives, our implementation checks for certain conditions before applying rule (2) from Figure 7. Specifically, given precondition ϕ and variables V accessed by shared statement S, our implementation applies rule (2) only when ϕ implies semantic conflict freedom on all variables in set V; otherwise, our implementation falls back on product construction (i.e., rule (6) from Figure 7). While this check fails rarely in practice, it is nonetheless useful for avoiding false positives.

7.1 Limitations

Our current prototype implementation has a few limitations:

Changes to method signature. While SafeMerge can handle renamed local variables, it currently does not automatically support renamed methods or methods with parameter reordering, introduction, or deletion. However, these features can be supported with additional annotations providing suitable mappings between renamed methods and parameters.

Analysis scope. SafeMerge analyzes one modified procedure at a time and assumes that external callees invoked by that procedure are semantically conflict-free.

Concurrency, termination, and exceptions. Neither our formalism nor our prototype implementation support sound reasoning in the presence of concurrency. Our soundness claims also rely on the assumption that none of the variants introduce non-terminating behavior. Finally, although exceptions can be conceptually desugared in our formalism, our implementation does not handle exceptional control flow.

8 EXPERIMENTAL EVALUATION

To assess the usefulness of the proposed method, we perform a series of three experiments. In our first experiment, we use SafeMerge to verify semantic conflict-freedom of merges collected from Github commit histories. In our second experiment, we run SafeMerge on erroneous merge candidates generated by `kdiff3`, a widely-used textual merge tool. Finally, in our third experiment, we assess the scalability of our method and the importance of various design choices. All experiments are performed on Quad-core Intel Xeon CPU with 2.4 GHz and 8 GB memory.

8.1 Evaluation on Github Merge Candidates

To perform our first experiment, we implemented a crawler that examines git commit histories and extracts *relevant* and *non-trivial* verification benchmarks that can be analyzed by our tool:

- ***Relevance:*** Since our main goal is to verify the correctness of automatically generated merges, we consider a benchmark to be irrelevant if it is known that the user overrode the automatic merge [11].
- ***Non-triviality:*** We consider a benchmark to be trivial if we can prove it to be semantically conflict-free using *purely syntactic* arguments. Specifically, we filter out benchmarks where (a) at least one of the variants is the same as the base, or (b) both variants are syntactically the same, or (c) the method does not involve externally visible side effects.

To see why benchmarks satisfying conditions (a)-(c) are trivially conflict-free, recall the definition of semantic conflict freedom (i.e., Definition 4.4): For case (a), if Variant A is the same as the Base

[11] For this purpose, we use kdiff3 as our automatic merge tool.

program, then the automatically generated merge will be the same as Variant B, ensuring that the antecedent of condition (1) is false and that the consequent of condition (2) holds in Definition 4.4. For case (b), if both variants are syntactically the same, then the automatically generated merge will be the same as both, meaning that the consequents of both (1) and (2) will hold in Definition 4.4. Finally, for case (c), all conditions vacuously hold since the *out* array used in Definition 4.4 is empty if the function does not have externally visible side effects.

Running this crawler on nine popular Java applications (namely, Elasticsearch, libGDX, iosched, kotlin, MPAndroidChart, okhttp, retrofit, RxJava, and the Spring Boot framework), we identified a total of 52 merge instances that satisfy the relevance and non-triviality criteria above. Each benchmark comes with a base program, two variants A and B, as well as the merge candidate that we extracted from the Github commit history. In the following discussion, we report the results of our evaluation on *all* 52 merge scenarios.

Main results. The results of our evaluation are presented in Table 1. We abbreviate "Elastic Search" as "Elastic" and "MPAndroidChart" as "AChart". For each benchmark, Table 1 shows the name of the application it is taken from (column "App"), the number of lines of code in the merge candidate ("LOC"), the running time of SafeMerge in seconds ("Time"), and the results produced by SafeMerge and `kdiff3`. Specifically, for SafeMerge, a checkmark (✓) indicates that it was able to verify semantic conflict-freedom, whereas ✗ means that it produced a warning. In the case of `kdiff3`, a checkmark indicates the absence of *syntactic* conflicts.

Table 1. Experimental results.

ID	App	LOC	Time (s)	SafeMerge	kdiff3
1	Elastic	18	0.05	✓	✗
2	Elastic	25	0.07	✓	✓
3	Elastic	101	0.20	✓	✓
4	Elastic	63	0.49	✓	✓
5	Elastic	90	4.45	✓	✓
6	Elastic	136	4.07	✓	✓
7	Elastic	15	2.09	✓	✓
8	Elastic	30	0.11	✗	✗
9	Elastic	25	0.09	✗	✗
10	Elastic	21	0.15	✗	✗
11	iosched	63	0.19	✓	✓
12	iosched	64	0.07	✓	✓
13	Kotlin	96	0.16	✗	✓
14	Kotlin	54	0.57	✓	✓
15	Kotlin	53	0.48	✓	✓
16	Kotlin	53	0.11	✓	✓
17	Kotlin	104	0.49	✓	✓
18	Kotlin	86	0.31	✓	✓
19	Kotlin	127	4.19	✓	✗
20	Kotlin	56	0.62	✓	✓
21	Kotlin	11	0.06	✓	✓
22	Kotlin	77	0.18	✓	✓
23	Kotlin	11	0.06	✓	✓
24	Kotlin	38	0.15	✓	✓
25	Kotlin	67	0.33	✓	✗
26	Kotlin	7	0.19	✗	✗
27	libGDX	30	0.12	✓	✓
28	libGDX	32	0.21	✓	✓
29	libGDX	71	0.16	✓	✓
30	AChart	47	0.44	✗	✓
31	AChart	66	0.17	✓	✓
32	AChart	109	0.16	✓	✓
33	AChart	44	0.10	✓	✓
34	AChart	62	0.16	✓	✓
35	AChart	43	0.11	✓	✗
36	AChart	35	0.23	✗	✗
37	AChart	37	0.39	✗	✗
38	okhttp	28	0.10	✗	✓
39	retrofit	66	1.67	✓	✓
40	retrofit	78	1.76	✓	✓
41	RxJava	28	0.20	✓	✓
42	Spring	107	0.12	✓	✓
43	Spring	77	0.23	✗	✗
44	Spring	82	0.15	✓	✓
45	Spring	81	0.21	✓	✓
46	Spring	44	0.15	✓	✗
47	Spring	37	0.30	✓	✗
48	Spring	42	0.07	✓	✓
49	Spring	36	0.06	✓	✓
50	Spring	64	0.20	✗	✓
51	Spring	13	0.09	✗	✗
52	Spring	20	0.05	✗	✓

As we can see from Table 1, SafeMerge is able to verify semantic conflict-freedom for 39 of the 52 benchmarks and reports a warning for the remaining 13. We manually inspected these thirteen benchmarks and found eleven instances of an actual semantic conflict (i.e., the merge candidate is indeed incorrect with respect to Definition 4.4). The remaining two warnings are false positives caused by imprecision in the dependence analysis and modeling of collections. In all, these results indicate that SafeMerge is quite precise, with a false positive rate around 15%. Furthermore, this experiment also corroborates that SafeMerge is practical, taking an average of 0.5 second to verify each benchmark.

Next, Table 2 compares the results produced by SafeMerge and `kdiff3` on the 52 benchmarks used in our evaluation. Note that this comparison is very relevant because the merge candidate in these benchmarks matches exactly the merge produced by `kdiff3` whenever it does not report a textual conflict. As shown in Table 2, 33 benchmarks are classified as conflict-free by both SafeMerge and `kdiff3`, meaning that SafeMerge can verify the

Table 2. Summary of differences between SafeMerge and kdiff3. "Count" denotes # of instances from Table 1.

SafeMerge	kdiff3	Count	Implication
✓	✓	33	Verified textual merge
✓	✗	6	Verified manual merge
✗	✓	5	Fail to verify textual merge
✗	✗	8	Fail to verify manual merge

Table 3. Results of our evaluation on merges generated by kdiff3.

Name	Description	Time (s)	Result
B1-kdiff3	Patch gets duplicated in merge	0.36	✗
B1-manual	Correct version of above	0.38	✓
B2-kdiff3	Semantically same, syntactically different patches	0.42	✗
B2-manual	Correct version of above	0.33	✓
B3-kdiff3	Inconsistent changes in assignment (conflict)	0.34	✗
B4-kdiff3	Interference between refactoring and insertion (conflict)	0.31	✗
B5-kdiff3	Interference between insertion and deletion (conflict)	0.30	✗
B6-kdiff3	One patch supercedes the other	0.32	✗
B6-manual	Correct version of above	0.29	✓
B7-kdiff3	Inconsistent patches due to off-by-one error (conflict)	0.29	✗

correctness of the textual merge generated by kdiff3 in these cases. For instance, the merge with ID 41 in Table 1 corresponds precisely to the example presented in Figure 3 from Section 2. Perhaps more interestingly, we find five benchmarks for which kdiff3 generates a textual merge that is semantically incorrect according to SafeMerge. Among these five instances, two correspond to the false positives discussed earlier, leaving us with three benchmarks where the merge generated by kdiff3 violates Definition 4.4 and should be further investigated by the developers.

As we can see from Table 2, there are fourteen benchmarks that are syntactically conflicting according to kdiff3 and were likely resolved manually by a developer. Among these, SafeMerge can verify the correctness of the merge candidate for six instances (spread over four different applications). Finally, there are eight cases where the manual merge cannot be verified by SafeMerge. While these examples indeed violate semantic conflict-freedom, they do not necessarily correspond to bugs (e.g., a developer might have intentionally discarded changes made by another developer). For example, in the merge with ID 36 from Table 1, both variants A and B weaken a predicate in two different ways by adding two and one additional disjuncts respectively [12]. However, the merge M only picks the weaker predicate from A, thereby effectively discarding some of the changes from variant B.

Overall, these results demonstrate that SafeMerge can *automatically* verify 75% of the merge scenarios that occur in the wild. Among the remaining 13 benchmarks that cannot be automatically verified, only 2 bencmarks (15%) are false positives. We believe these results indicate that SafeMerge is useful for analyzing semantic conflict-freedom in real-world merge instances.

8.2 Evaluation on Erroneous Merge Candidates

In our second experiment, we explore whether SafeMerge is able to pinpoint erroneous merges generated by kdiff3. To perform this experiment, we consider base program with ID = 25 from Table 1 and generate variants by performing various kinds of mutations to the base program. Specifically, we design pairs of mutations that cause kdiff3 to generate buggy merge candidates.

[12]Merge commit https://github.com/PhilJay/MPAndroidChart/commit/9531ba69895cd64fce48038ffd8df2543eeea1d2

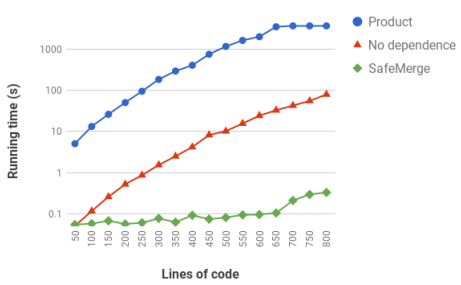

Fig. 9. Lines of code vs. running time.

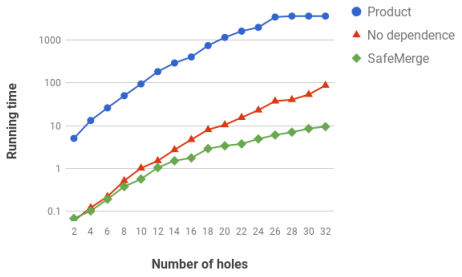

Fig. 10. Number of holes (edits) vs. running time. Lines of code varies between 50 and 800.

The results of this experiment are summarized in Table 3, where the column labeled "Description" summarizes the nature of the mutation. For each pair of variants that are semantically conflict-free, the version named -KDIFF3 shows the incorrect merge generated by `kdiff3`, where as the one labeled -MANUAL shows the correct merge that we generated manually. For benchmarks that are semantically conflicting, we only provide results for the incorrect merge generated by `kdiff3` since a correct merge simply does not exist.

The results from Table 3 complement those from Section 8.1 and provide further evidence that a widely-used merge tool like `kdiff3` can generate erroneous merges and that these buggy merges can be detected by our proposed technique. This experiment also demonstrates that SAFEMERGE can verify conflict-freedom in the manually constructed correct merges.

8.3 Evaluation of Scalability and Design Choices

To assess the scalability of the proposed technique, we performed a third experiment in which we compare the running time of SAFEMERGE against the number of lines of code and number of edits. To perform this experiment, we start with an existing benchmark from the SAFEMERGE test suite and increase the number of lines of code using loop unrolling. We also vary the number of edits by injecting a modification in the loop body. This way, the number of holes in the shared program increases with each loop unrolling.

To evaluate the benefits of the various design choices that we adopt in this paper, we also compare SAFEMERGE with two variants of itself. In one variant, namely *Product*, we model the shared program using a single hole, so each edit corresponds to one of the program versions. Essentially, this method computes the product of the four program versions using the rules from Figure 8 and allows us to assess the benefits of representing program versions as edits applied to a shared program. In another variant called *No dependence*, we do not abstract away shared program fragments using uninterpreted functions and analyze them by constructing a 4-way product. However, we still combine reasoning from different product programs in a compositional way.

Figure 9 compares the running time of SAFEMERGE against these two variants as we vary the number of lines of code but *not* the number of edits. Observe that the y-axis is shown in log scale. As we can see from this plot, SAFEMERGE scales quite well and analyzes each benchmark in under a second. In contrast, the running time of *Product* grows exponentially in the lines of code. As expected, the *No dependence* variant is better than *Product* but significantly worse than SAFEMERGE.

Next, Figure 10 compares the running time of SAFEMERGE against *Product* and *No dependence* as we vary *both* the number of lines of code and the number of edits. Specifically, a benchmark containing n holes contains $25n$ lines of code, and the y-axis shows the running time of each variant

in log scale. As expected, SafeMerge is more sensitive to the number holes than it is to the number of lines of code because it abstracts away shared program fragments. However, SafeMerge still significantly outperforms both *Product* and *No Dependence*. In particular, for a program with 32 edits and 800 lines of code, SafeMerge can verify semantic conflict freedom in approximately 10 seconds, while *No Dependence* takes approximately 100 seconds and *Product* times out.

In summary, this experiment shows that SafeMerge scales well as we vary the lines of code and that its running time is still feasible when program variants perform over 30 modifications to the base program in this example. This experiments also corroborates the practical importance of representing program versions as edits applied to a shared program as well as the advantage of abstracting away shared program fragments using uninterpreted functions.

9 RELATED WORK

In this section, we compare our technique with prior work on program merging and relational verification.

Structure-aware merge. Most algorithms for program merging are textual in nature, hardly ever formally described [Khanna et al. 2007], and without semantic guarantees. To improve on this situation, previous work has proposed *structured* and *semi-structured* merge techniques to better resolve merge conflicts. For example, `FSTMerge` [Apel et al. 2011] uses syntactic structure to resolve conflicts between AST nodes that can be reordered (such as method definitions), but it falls back on unstructured textual merge for other kinds of nodes. Follow-up work on `JDime` [Apel et al. 2012; Lebetaenich et al. 2015] improves the poor performance of structure-based merging by using textual-based mode (fast) as long as no conflicts are detected, but switches to structure-based mode in the presence of conflicts. Asenav et al. present another merge algorithm for better version control of structured data [Asenov et al. 2017]. While their approach relies on line-based differencing, it also utilizes the underlying tree structure to more accurately identify changes and synthesize a higher-quality three-way merge. While alll of these techniques attempt to do a better job at merging ASTs compared to purely textual merge, none of them can guarantee semantic conflict freedom.

Semantics-aware merge. Our work is inspired by earlier work on *program integration*, which originated with the HPR algorithm [Horwitz et al. 1989] for checking *non-interference* and generating valid merges. The HPR algorithm was later refined by the work of Yang et al. [Yang et al. 1990], which is one of the first attempts to incorporate semantics for merge generation. In that context, the notion of conflict-freedom is parameterized by a classification of nodes of the variants as *unchanged* such that the backward slices of unchanged nodes in the two variants are equivalent modulo a semantic correspondence. Thus, their classification algorithm is parameterized by a semantic *congruence* relation. Our approach tackles the slightly different *merge verification* (rather then *merge generation*) problem, but improves on these prior techniques in several dimensions: First, we do not require annotations to map statements across the different versions — this information is computed automatically using our edit generation algorithm (Sec 6). Second, we show how to formulate conflict freedom directly with verification conditions and assertion checking. Finally, our approach performs precise, compositional reasoning about edits by combining lightweight dependence analysis with relational reasoning using product programs.

Empirical studies. There is a rich literature on empirically studying version control systems and merge conflicts in the wild. For example, Mehdi et al. measure programmer effort required to use various *textual* merge tools [Mehdi et al. 2014]. They study six open-source projects from Github and compare scenarios where one merge technique successfully generates a candidate merge while another one reports a conflict. In a similar vein, Brun et al. study nine open-source systems totaling over 3 million lines of code and study the frequency of conflicts identified by textual merge tools. Their findings show that 16% of merges require manual effort to resolve conflicts and that 33%

of *textually-conflict-free* merges do in fact contain semantic conflicts [Brun et al. 2013]. Recent work by Cavalcanti et al. [Cavalcanti et al. 2017] performs a large-scale empirical study comparing *structure-aware* merge tools. They study 30,000 merges from 50 open source projects and study both false positives (i.e., spurious conflicts) as well as false negatives (i.e., unreported conflicts).

Early conflict detection. Motivated by the difficulty of fixing merge conflicts, several prior papers propose methods for *early conflict detection* [Brun et al. 2013; Estler et al. 2013; Guimarães and Silva 2012]. For example, Brun et al. introduce *speculative analysis* for performing version control system operations on clones of the program in the background [Brun et al. 2013]. Similarly, Guimaraes et al. propose continuous merging inside an integrated development environment to enable early conflict detection and conduct an empirical study to assess the impact of this approach [Guimarães and Silva 2012]. Estler et al. propose CloudStudio, a system that operates continuously in the background and shares one developer's changes with other developers [Estler et al. 2013]. All of these techniques are intended for identifying textual conflicts early and do not address scenarios where a bug is introduced by the automatic merge tool.

Relational verification. Verification of conflict freedom is related to a line of work on relational program logics [Benton 2004; Sousa and Dillig 2016; Yang 2007] and product programs [Barthe et al. 2011, 2013; Lahiri et al. 2013; Zaks and Pnueli 2008]. For instance, Benton's Relational Hoare Logic (RHL) [Benton 2004] allows proving equivalence between a pair of structurally similar programs. Sousa and Dillig generalize Benton's work by developing Cartesian Hoare Logic, which is used for proving k-safety properties that reqiure the absence of a bad interaction between different runs of a given program [Sousa and Dillig 2016]. Barthe et al. propose another technique for relational verification using *product programs* [Barthe et al. 2011, 2013] and apply their technique to relational properties, such as equivalence and 2-safety [Terauchi and Aiken 2005]. In this work, we build on the notion of product programs used in prior work [Barthe et al. 2011, 2013; Zaks and Pnueli 2008]. However, rather than constructing a monolithic product of the four program version, we construct *mini-products* for each edit in a novel way.

Cross-version program analysis. Prior work on comparing closely related programs versions include *regression verification* that checks semantic equivalence using uninterpreted function abstraction of equivalent callees [Felsing et al. 2014; Godlin and Strichman 2008; Lahiri et al. 2012], mutual summaries [Hawblitzel et al. 2013; Wood et al. 2017], relational invariant inference to prove differential properties [Lahiri et al. 2013] and verification modulo versions [Logozzo et al. 2014]. Other approaches include static analysis for abstract differencing [Jackson and Ladd 1994; Partush and Yahav 2014], symbolic execution for verifying assertion-equivalence [Ramos and Engler 2011] and *differential symbolic execution* to summarize differences [Person et al. 2008]. Work on differential assertion checking [Lahiri et al. 2013] uses interprocedural product programs and relational invariant inference to ensure that a change preserves non-failing behaviors with respect to a set of assertions. However, we do not require assertions and verify a more complex property involving four different programs.

10 CONCLUSION AND FUTURE WORK

Motivated by the abundance of bugs that are introduced in the automatic merge process, we have introduced the *merge verification* problem for proving that four versions of a given program satisfy a semantic notion of *conflict freedom*. Even though proving semantic conflict freedom often requires sophisticated static analysis, analysis efficiency is an important concern in this context due to the interactive nature of automatic program merging. We address this problem by introducing an efficient and compositional verification algorithm that leverages shared program parts between the four versions. In particular, our method intertwines deep relational reasoning about program

$$\frac{}{\sigma \vdash \text{skip} \Downarrow \sigma} \qquad \frac{\sigma \vdash e \Downarrow c \quad \sigma' = \sigma[(x,0) \mapsto c]}{\sigma \vdash x := e \Downarrow \sigma'}$$

$$\frac{\sigma \vdash e_1 \Downarrow c_1 \quad \sigma \vdash e_2 \Downarrow c_2 \quad \sigma' = \sigma[(x,c_1) \mapsto c_2]}{\sigma \vdash x[e_1] := e_2 \Downarrow \sigma'} \qquad \frac{\sigma[(x,0)] = c}{\sigma \vdash out(x) \Downarrow \sigma}$$

$$\frac{\sigma \vdash S_1 \Downarrow \sigma_1 \quad \sigma_1 \vdash S_2 \Downarrow \sigma_2}{\sigma \vdash S_1; S_2 \Downarrow \sigma_2} \qquad \frac{\sigma \vdash C \Downarrow \text{true} \quad \sigma \vdash S_1 \Downarrow \sigma_1}{\sigma \vdash C \ ? \ \{S_1\} : \{S_2\} \Downarrow \sigma_1}$$

$$\frac{\sigma \vdash C \Downarrow \text{false} \quad \sigma \vdash S_2 \Downarrow \sigma_2}{\sigma \vdash C \ ? \ \{S_1\} : \{S_2\} \Downarrow \sigma_2} \qquad \frac{\sigma \vdash C \Downarrow \text{false}}{\sigma \vdash \text{while}(C)\ \{S\} \Downarrow \sigma}$$

$$\frac{\sigma \vdash C \Downarrow \text{true} \quad \sigma \vdash S \Downarrow \sigma_1 \quad \sigma_1 \vdash \text{while}(C)\ \{S\} \Downarrow \sigma_2}{\sigma \vdash \text{while}(C)\ \{S\} \Downarrow \sigma_2}$$

Fig. 11. Operational semantics

edits with lightweight summarization of the shared fragments to achieve a good trade-off between analysis precision and efficiency.

We have implemented the proposed approach in a tool called SafeMerge and performed an extensive evaluation on real-world merge scenarios obtained from Github. Our evaluation shows the proposed approach remedies many of the shortcomings associated with existing syntactic merge tools. Furthermore, our approach is both efficient and precise, with an overall false positive rate of 15% across the 52 Github benchmarks.

More generally, we view this work as a first step towards precise, semantics-aware *merge synthesis*. Specifically, we plan to explore synthesis techniques that can automatically generate correct-by-construction 3-way program merges. Since correct merge candidates should obey semantic conflict freedom, the verification algorithm proposed in this paper is necessarily a key ingredient of such semantics-aware merge synthesis tools.

Appendix A: Operational Semantics

Figure 11 shows the operational semantics of the language from Figure 5. Recall that σ maps (variable, index) pairs to values, and we view scalar variables as arrays with a single valid index at 0. Since the semantics of expressions is completely standard, we do not show them here. However, one important point worth noting is the semantics of expressions involving array reads:

$$\frac{\sigma \vdash e \Downarrow c \quad (a,c) \in dom(\sigma)}{\sigma \vdash a[e] \Downarrow \sigma[(a,c)]} \qquad \frac{\sigma \vdash e \Downarrow c \quad (a,c) \notin dom(\sigma)}{\sigma \vdash a[e] \Downarrow \bot}$$

In other words, reads from locations that have not been initialized yield a special constant $\bot$.

Appendix B: Soundness of Product

Here, we provide a proof of Theorem 5.4. The proof is by structural induction over the product construction rules given in Figure 8. Since the two directions of the proof are completely symmetric, we only prove one direction. Note that the base case is trivial because $\vdash \mathcal{S} \rightsquigarrow \mathcal{S}$.

Rule 1. Suppose $\sigma \vdash A \Downarrow \sigma'$ and $\sigma' \vdash \mathcal{S}_1; \mathcal{S}_2; \ldots; \mathcal{S}_n \Downarrow \sigma''$. By the premise of the proof rule and the inductive hypothesis, we have $\sigma' \vdash \mathcal{S} \Downarrow \sigma''$. Thus, $\sigma \vdash A; \mathcal{S} \Downarrow \sigma''$.

Rule 2. Suppose $\sigma \vdash (C \ ? \ \{\mathcal{S}_t\} : \{\mathcal{S}_e\}); \mathcal{S}_1; \mathcal{S}_2; \ldots; \mathcal{S}_n \Downarrow \sigma''$. Without loss of generality, suppose $\sigma \vdash C \Downarrow \mathit{true}$, and suppose $\sigma \vdash \mathcal{S}_t; \mathcal{S}_1 \Downarrow \sigma'$, so $\sigma' \vdash \mathcal{S}_2; \ldots; \mathcal{S}_n \Downarrow \sigma''$. By the first premise of the proof rule and the inductive hypothesis, we have $\sigma \vdash \mathcal{S}' \Downarrow \sigma''$. Hence, $\sigma \vdash C \ ? \ \{\mathcal{S}'\} : \{\mathcal{S}''\} \Downarrow \sigma''$.

Rule 3. Let $\mathcal{S}_x = \text{while}(C_1) \ \{\mathcal{S}_{B_1}\}; \mathcal{S}_1$. Suppose we have $\sigma \vdash \mathcal{S}_x; \mathcal{S}_2; \ldots; \mathcal{S}_n \Downarrow \sigma'$. Suppose there is exists $\mathcal{S}_i$ that satisfies first premise of the proof rule. Observe that $\mathcal{S}_1; \mathcal{S}_2; \ldots; \mathcal{S}_n;$ is semantically equivalent to $\mathcal{S}_n; \mathcal{S}_2; \ldots; \mathcal{S}_{n-1}; \mathcal{S}_1;$ as long as $\mathcal{S}_1$, $\mathcal{S}_n$, do not share variables between them and also with $\mathcal{S}_2 \ldots \mathcal{S}_{n-1}$. Since $\mathcal{S}_x$ and $\mathcal{S}_i$ have no shared variables between them and with any other program $\mathcal{S}_j$ different than $\mathcal{S}_x$ and $\mathcal{S}_i$, we have

$$\sigma \vdash \mathcal{S}_i; \mathcal{S}_2; \ldots; \mathcal{S}_{i-1}; \mathcal{S}_{i+1}; \ldots \mathcal{S}_n; \text{while}(C_1) \ \{\mathcal{S}_{B_1}\}; \mathcal{S}_1 \Downarrow \sigma'$$

Then, by the premise of the proof rule and the inductive hypothesis, we have $\sigma \vdash \mathcal{S} \Downarrow \sigma'$.

Rule 4. Suppose we have $\sigma \vdash \mathcal{S}_1; \mathcal{S}_2; \ldots; \mathcal{S}_n \Downarrow \sigma''$ where each $\mathcal{S}_i$ is of the form $\text{while}(C_i) \ \{\mathcal{S}_{B_i}\}; \mathcal{S}'_i$. By the same reason as in Rule 3. we can move any loop in each $\mathcal{S}_i$ to the beginning as they don't share any variable with any other $\mathcal{S}_j$. That is, considering $H = \mathcal{S}_1; \ldots; \mathcal{S}_o$ be the set of programs satisfying the second premise we have

$$\sigma \vdash \text{while}(C_1) \ \{\mathcal{S}_{B_1}\}; \ldots \text{while}(C_o) \ \{\mathcal{S}_{B_o}\} \Downarrow \sigma'$$

and considering $\mathcal{S}_{o+1}; \ldots \mathcal{S}n$ a sequence of the original programs excluding the ones in H we have

$$\sigma' \vdash \mathcal{S}'_1; \ldots \mathcal{S}'_o; \mathcal{S}_{o+1}; \ldots \mathcal{S}n \Downarrow \sigma''$$

Then, by the last premises of the proof rule and the inductive hypothesis, we have that $\sigma \vdash \mathcal{S}'; \mathcal{S}'' \Downarrow \sigma''$.

Rule 5. Suppose we have

$$\sigma \vdash \text{while}(C_1) \ \{\mathcal{S}_1\}; \text{while}(C_2) \ \{\mathcal{S}_2\}; \mathcal{S}_3; \ldots; \mathcal{S}_n \Downarrow \sigma'$$

Let W' be the loop $\text{while}(C_1 \wedge C_2) \ \{\mathcal{S}_1; \mathcal{S}_2\}$. Since C_1, C_2 and $\mathcal{S}_1, \mathcal{S}_2$ have disjoint sets of variables, the program fragment $\text{while}(C_1) \ \{\mathcal{S}_1\}; \text{while}(C_2) \ \{\mathcal{S}_2\}$ is semantically equivalent to $W'; R$ (where R comes from the third line of the proof rule). Hence, we have $\sigma \vdash W'; R; \mathcal{S}_3; \ldots; \mathcal{S}_n \Downarrow \sigma'$. By the first premise of the proof rule and the inductive hypothesis, if $\sigma_0 \vdash \mathcal{S}_1; \mathcal{S}_2 \Downarrow \sigma_1$ for any σ_0, σ_1, then $\sigma_0 \vdash \mathcal{S} \Downarrow \sigma_1$. Thus, $\sigma \vdash W' \Downarrow \sigma^*$ implies $\sigma \vdash W \Downarrow \sigma^*$, which in turn implies $\sigma \vdash W; R; \mathcal{S}_3; \ldots; \mathcal{S}_n \Downarrow \sigma'$. By the last premise of the proof rule and the inductive hypothesis, we know $\sigma \vdash \mathcal{S}' : \sigma'$; hence, the property holds.

Appendix C: Proof of Soundness of Relational Post-conditions

The proof is by structural induction on $\hat{\mathcal{S}}$.

Case 1. $\hat{\mathcal{S}} = [\cdot]$, and the edits are $\mathcal{S}_1, \ldots, \mathcal{S}_4$. In this case, Figure 7 constructs the relational post-condition by first computing the product program $\mathcal{S}$ as $\mathcal{S}_1[V_1/V] \circledast \ldots \circledast \mathcal{S}[V_4/4]$ and then computing the standard post-condition of $\mathcal{S}$. By Theorem 5.4, we have $\sigma \vdash \mathcal{S} \Downarrow \sigma'$ iff $\sigma \vdash \mathcal{S}_1[V_1/V] \circledast \ldots \circledast \mathcal{S}[V_4/4] \Downarrow \sigma'$. Furthermore, by the correctness of *post* operator, we know that $\{\varphi\}\mathcal{S}\{\varphi'\}$ is a valid Hoare triple. This implies $\{\varphi\}\mathcal{S}_1[V_1/V]; \ldots; \mathcal{S}_4[V_4/V]\{\varphi'\}$ is also a valid Hoare triple.

Case 2. $\hat{\mathcal{S}} = \mathcal{S}$ (i.e., $\hat{\mathcal{S}}$ does not contain holes). By the second rule in Figure 7, we know that $\{\varphi\}\mathcal{S}_1; \ldots; \mathcal{S}_n\{\varphi'\}$ is a valid Hoare triple. Now, consider any valuation σ satisfying φ. By the correctness of the Hoare triple, if $\sigma \vdash \mathcal{S}_1; \ldots; \mathcal{S}_n \Downarrow \sigma'$, we know that σ' also satisfies φ'. Now, recall that $\mathcal{S}_1; \ldots; \mathcal{S}_n$ contains uninterpreted functions, and we assume that $F(\vec{x})$ can return any value, as long as it returns something consistent for the same input values. Let Σ represent the set of all valuations σ_i such that $\sigma \vdash \mathcal{S}_1; \ldots; \mathcal{S}_n \Downarrow \sigma_i$. By the correctness of the Hoare triple, we know that any $\sigma_i \in \Sigma$ satisfies φ'. Assuming the correctness of the mod and dependence analysis, for any valuation σ such that $\sigma \vdash \mathcal{S}[V_1/V]; \ldots; \mathcal{S}[V_4/V] \Downarrow \sigma'$, we know that $\sigma' \in \Sigma$. Since all valuations in Σ satisfy φ', this implies σ' also satisfies φ'. Thus, $\{\varphi\}\mathcal{S}[V_1/V]; \ldots; \mathcal{S}[V_4/V]\{\varphi'\}$ is also a valid Hoare triple.

Case 3. $\hat{\mathcal{S}} = \hat{\mathcal{S}}_1; \hat{\mathcal{S}}_2$. Let $\vec{\Delta}_A$ denote the prefix of $\vec{\Delta}$ that is used for filling holes in $\hat{\mathcal{S}}_1$, and $\vec{\Delta}_B$ denote the prefix of $\vec{\Delta}_1$ that is used for filling holes in $\hat{\mathcal{S}}_2$. By the premise of the third rule and inductive hypothesis, we have

$$\{\varphi\}(\hat{\mathcal{S}}_1[\Delta_{A1}])[V_1/V]); \ldots; (\hat{\mathcal{S}}_1[\Delta_{A4}])[V_4/V])\{\varphi_1\}$$

as well as

$$\{\varphi_1\}(\hat{\mathcal{S}}_2[\Delta_{B1}])[V_1/V]); \ldots; (\hat{\mathcal{S}}_2[\Delta_{B4}])[V_4/V])\{\varphi_2\}$$

Using these and the standard Hoare rule for composition, we can conclude:

$$\begin{array}{lll} \{\varphi\} & (\hat{\mathcal{S}}_1[\Delta_{A1}])[V_1/V]); \ldots; (\hat{\mathcal{S}}_1[\Delta_{A4}])[V_4/V]); & \\ & (\hat{\mathcal{S}}_2[\Delta_{B1}])[V_1/V]); \ldots; (\hat{\mathcal{S}}_2[\Delta_{B4}])[V_4/V]) & \{\varphi_2\} \end{array}$$

Since we can commute statements over different variables, this implies:

$$\begin{array}{lll} \{\varphi\} & (\hat{\mathcal{S}}_1[\Delta_{A1}]; \hat{\mathcal{S}}_2[\Delta_{B1}])[V_1/V]); \ldots; & \\ & (\hat{\mathcal{S}}_1[\Delta_{A4}]; \hat{\mathcal{S}}_2[\Delta_{B4}])[V_4/V]) & \{\varphi_2\} \end{array}$$

Next, using the fact that $\hat{\mathcal{S}}[\Delta_i] = (\hat{\mathcal{S}}_1; \hat{\mathcal{S}}_2)[\Delta_i] = \hat{\mathcal{S}}_1[\Delta_{Ai}]; \hat{\mathcal{S}}_2[\Delta_{Bi}]$, we can conclude:

$$\{\varphi\}(\hat{\mathcal{S}}[\Delta_1])[V_1/V]); \ldots; (\hat{\mathcal{S}}[\Delta_4])[V_4/V])\{\varphi_2\}$$

Case 4. $\hat{\mathcal{S}} = C\ ?\ \{\hat{\mathcal{S}}_1\} : \{\hat{\mathcal{S}}_2\}$. Let $\vec{\Delta}_A, \vec{\Delta}_B$ denote the prefixes of $\vec{\Delta}, \vec{\Delta}_1$ that is used for filling holes in $\hat{\mathcal{S}}_1$ and $\hat{\mathcal{S}}_2$ respectively. Also, let C_i denote $C[V_i/V]$. By the first premise of rule 4 from Figure 7 and the inductive hypothesis, we have:

$$\{\varphi \wedge C_1\}\ \ (\hat{\mathcal{S}}_1[\Delta_{A1}])[V_1/V]); \ldots; (\hat{\mathcal{S}}_1[\Delta_{A4}])[V_4/V])\ \ \{\varphi_1\}$$

Now, using the second premise and the inductive hypothesis, we also have:

$$\{\varphi \wedge \neg C_1\}\ \ (\hat{\mathcal{S}}_2[\Delta_{B1}])[V_1/V]); \ldots; (\hat{\mathcal{S}}_2[\Delta_{B4}])[V_4/V])\ \ \{\varphi_2\}$$

Using these two facts and the standard Hoare logic rule for if statements, we get:

$$\begin{array}{lll} \{\varphi\} & C_1?(\hat{\mathcal{S}}_1[\Delta_{A1}])[V_1/V]); \ldots; (\hat{\mathcal{S}}_1[\Delta_{A4}])[V_4/V]) : & \\ & (\hat{\mathcal{S}}_2[\Delta_{B1}])[V_1/V]); \ldots; (\hat{\mathcal{S}}_2[\Delta_{B4}])[V_4/V]) & \{\varphi_1\} \end{array}$$

Now, since φ logically entails $\bigwedge_{i,j} C_i \leftrightarrow C_j$, the statement above is equivalent to:

$$\begin{array}{l} C_1 \ ? \ \{(\hat{\mathcal{S}}_1[\Delta_{A1}])[V_1/V]\} : \{(\hat{\mathcal{S}}_2[\Delta_{B1}])[V_1/V]\}; \\ \ldots \\ C_4 \ ? \ \{(\hat{\mathcal{S}}_1[\Delta_{A4}])[V_4/V]\} : \{(\hat{\mathcal{S}}_2[\Delta_{B4}])[V_4/V]\}; \end{array}$$

Next, using the fact that $\hat{\mathcal{S}}[\Delta_i] = (C \ ? \ \{\hat{\mathcal{S}}_1\} : \{\hat{\mathcal{S}}_2\})[\Delta_i] = C \ ? \ \{\hat{\mathcal{S}}_1[\Delta_{Ai}]\} : \{\hat{\mathcal{S}}_2[\vec{\Delta}_{Bi}]\}$, we can conclude:

$$\begin{array}{lll} \{\varphi\} & ((C_1 \ ? \ \{\hat{\mathcal{S}}_1\} : \{\hat{\mathcal{S}}_2\})[\Delta_1])[V_1/V]; \ldots; & \\ & ((C_4 \ ? \ \{\hat{\mathcal{S}}_1\} : \{\hat{\mathcal{S}}_2\}))[\Delta_4])[V_4/V] & \{\varphi'\} \end{array}$$

Case 5. $\hat{\mathcal{S}} = \text{while}(C) \ \{\hat{\mathcal{S}}\}$. As in case (4), let C_i denote $C[V_i/V]$. From the premise of rule (5) of Figure 7 and the inductive hypothesis, we know:

$$\{\mathcal{I}\} \ \ (\hat{\mathcal{S}}[\Delta_1])[V_1/V]; \ldots (\hat{\mathcal{S}}[\Delta_4])[V_4/V] \ \ \{\mathcal{I}\}$$

Since we also have $\varphi \models \mathcal{I}$ from the premise, this implies:

$$\{\varphi\} \ \ \text{while}(C_1) \ \{(\hat{\mathcal{S}}[\Delta_1])[V_1/V]; \ldots (\hat{\mathcal{S}}[\Delta_4])[V_4/V]\} \ \ \{\mathcal{I} \wedge \neg C_1\}$$

Next, since we can commute statements over different variables and $\mathcal{I}$ implies $\bigwedge_{ij} C[V_i/V] \leftrightarrow C[V_j/V]$, we can conclude:

$$\begin{array}{lll} \{\varphi\} & \text{while}(C_1) \ \{(\hat{\mathcal{S}}[\Delta_1])[V_1/V]\}; \ldots; & \\ & \text{while}(C_4) \ \{(\hat{\mathcal{S}}[\Delta_4])[V_4/V]\} & \{\mathcal{I} \wedge \neg C_1\} \end{array}$$

Finally, because the loop $\text{while}(C_i) \ \{\hat{\mathcal{S}}[\Delta_i]\}$ is the same as $(\text{while}(C_i) \ \{\hat{\mathcal{S}}\})[\Delta_i]$, we have:

$$\begin{array}{lll} \{\varphi\} & (\text{while}(C_1) \ \{\hat{\mathcal{S}}[V_1/V]\})[\Delta_1]; \ldots; & \\ & (\text{while}(C_4) \ \{\hat{\mathcal{S}}[V_4/V]\})[\Delta_4] & \{\mathcal{I} \wedge \neg C_1\} \end{array}$$

Case 6. First, assuming the soundness of the standard *post* operator, we have $\{\varphi\}\mathcal{S}\{post(\mathcal{S}, \varphi)\}$. Using the premise of the proof rule and Theorem 5.4, we obtain:

$$\{\varphi\} \ \ (\hat{\mathcal{S}}[\Delta_1^1])[V_1/V]; \ldots (\hat{\mathcal{S}}[\Delta_4^1])[V_4/V] \ \ \{post(\mathcal{S}, \varphi)\}$$

Since Δ_i^1 is the prefix of Δ_i that contains as many holes as $\hat{\mathcal{S}}$, we also know $\hat{\mathcal{S}}[\Delta_i^1] = \hat{\mathcal{S}}[\Delta_i]$. Thus, we get:

$$\{\varphi\} \ \ (\hat{\mathcal{S}}[\Delta_1])[V_1/V]; \ldots (\hat{\mathcal{S}}[\Delta_4])[V_4/V] \ \ \{post(\mathcal{S}, \varphi)\}$$

Appendix D: Soundness of *n*-way Diff Algorithm

Theorem 6.1 follows directly from the following two lemmas:

Lemma 10.1. *If* $|\Delta| = \text{numHoles}(\hat{\Delta})$, *then* Compose *ensures the following post-conditions:*

- $|\Delta'| = |\hat{\Delta}|$
- *For any* $\hat{\mathcal{S}}$ *s.t.* $\text{numHoles}(\hat{\mathcal{S}}) = |\hat{\Delta}|$, $(\hat{\mathcal{S}}[\hat{\Delta}])[\Delta] = \hat{\mathcal{S}}[\Delta']$

Proof. Consider the two postconditions of Compose. For the branch $\hat{\Delta} = [\ ]$, it is easy to see that $\Delta' = \hat{\Delta} = [\ ]$ and thus $|\Delta'| = |\hat{\Delta}|$. For any $\hat{\mathcal{S}}$ with 0 holes, applying any edits gets back $\hat{\mathcal{S}}$, satisfying the second postcondition.

For the branch $\text{head}(\hat{\Delta}) = [\cdot]$, we know $\text{numHoles}(\text{tail}(\hat{\Delta})) = \text{numHoles}(\hat{\Delta}) - 1 = |\text{tail}(\Delta)|$ (given the precondition), which satisfies the precondition of Compose at line 21. The first postcondition of the recursive call to Compose implies that size of the return value ($|\Delta'|$) equals $|\text{head}(\Delta)| + |\text{tail}(\hat{\Delta})| = 1 + |\hat{\Delta}| - 1 = |\hat{\Delta}|$. Now consider a $\hat{\mathcal{S}}$ such that $\text{numHoles}(\hat{\mathcal{S}}) = |\hat{\Delta}|$. Let Δ'' be the return from

the recursive call to Compose. Then $\hat{\mathcal{S}}[\Delta'] = \hat{\mathcal{S}}[\text{head}(\Delta) :: \Delta''] = (\hat{\mathcal{S}}[\text{head}(\Delta)])[\Delta'']$ (by definition of applying an edit). Since $\text{numHoles}(\hat{\mathcal{S}}[\text{head}(\Delta)]) = \text{numHoles}(\hat{\mathcal{S}}) - 1 = |\text{tail}(\hat{\Delta})|$, we know that $(\hat{\mathcal{S}}[\text{head}(\Delta)])[\Delta''] = ((\hat{\mathcal{S}}[\text{head}(\Delta)])[\text{tail}(\hat{\Delta})])[\text{tail}(\Delta)]$ (from the second postcondition of the recursive call). Since $\text{head}(\hat{\Delta}) = [\cdot]$ in this branch, $(\hat{\mathcal{S}}[\text{head}(\Delta)])[\text{tail}(\hat{\Delta})] = (\hat{\mathcal{S}}[[\cdot] :: \text{tail}(\hat{\Delta})])[\text{head}(\Delta)] = (\hat{\mathcal{S}}[\hat{\Delta}])[\text{head}(\Delta)]$. This follows from the fact that applying $\text{head}(\Delta)$ to the first hole in $\hat{\mathcal{S}}$ followed by applying $\text{tail}(\hat{\Delta})$ is identical to applying a hole in the first hole in $\hat{\mathcal{S}}$ followed by applying $\text{tail}(\hat{\Delta})$, followed by applying $\text{head}(\Delta)$ which applies it to the first hole in $\hat{\mathcal{S}}$. Further, $((\hat{\mathcal{S}}[\hat{\Delta}])[\text{head}(\Delta)])[\text{tail}(\Delta)] = (\hat{\mathcal{S}}[\hat{\Delta}])[\Delta]$ by the rule of applying edits, which proves this postcondition.

For the branch $\text{head}(\hat{\Delta}) \neq [\cdot]$, we know $\text{numHoles}(\text{tail}(\hat{\Delta})) = \text{numHoles}(\hat{\Delta})$. This along with the precondition of Compose establishes the preconditon to the call to Compose at line 22. Let Δ'' denote the return of the recursive call to Compose. The recursive call ensures that $|\Delta''| = |\text{tail}(\hat{\Delta})| = |\hat{\Delta}| - 1$. Thus $|\Delta'| = |\text{head}(\hat{\Delta}) :: \Delta''| = |\hat{\Delta}|$, which establishes the first postcondition. Now consider a $\hat{\mathcal{S}}$ such that $\text{numHoles}(\hat{\mathcal{S}}) = |\hat{\Delta}|$. Then $\hat{\mathcal{S}}[\Delta'] = \hat{\mathcal{S}}[\text{head}(\hat{\Delta}) :: \Delta''] = (\hat{\mathcal{S}}[\text{head}(\hat{\Delta})])[\Delta'']$. Since $\text{numHoles}(\hat{\mathcal{S}}[\text{head}(\hat{\Delta})]) = \text{numHoles}(\hat{\mathcal{S}}) - 1 = |\text{tail}(\hat{\Delta})|$, we know that $(\hat{\mathcal{S}}[\text{head}(\hat{\Delta})])[\Delta''] = ((\hat{\mathcal{S}}[\text{head}(\hat{\Delta})])[\text{tail}(\hat{\Delta})])[\Delta]$ (from the second postcondition of the recursive call), which simplifies to $(\hat{\mathcal{S}}[\text{head}(\hat{\Delta}) :: \text{tail}(\hat{\Delta})])[\Delta] = (\hat{\mathcal{S}}[\hat{\Delta}])[\Delta]$ by the property of applying an edit. □

Lemma 10.2. *If $|\Delta_i| = \text{numHoles}(\hat{\mathcal{S}})$ for all $i \in [1, \ldots, k]$ and Diff2 satisfies the specification provided in Algorithm 3, then GenEdit ensures the following post-conditions:*

- $|\Delta'_i| = \text{numHoles}(\hat{\mathcal{S}}')$ *for* $i \in [1, \ldots, k+1]$
- $\hat{\mathcal{S}}'[\Delta'_{k+1}] = \mathcal{S}$ *and* $\hat{\mathcal{S}}'[\Delta'_i] = \hat{\mathcal{S}}[\Delta_i]$ *for* $i \in [1, \ldots, k]$

Proof. First, the precondition $|\Delta_i| = \text{numHoles}(\hat{\Delta})$ of Compose in line 12 is satisfied from the precondition $|\Delta_i| = \text{numHoles}(\hat{\mathcal{S}})$ of GenEdit and the second postcondition $\text{numHoles}(\hat{\Delta}) = \text{numHoles}(\hat{\mathcal{S}})$ of Diff2.

Now, consider the postcondition $|\Delta'_i| = \text{numHoles}(\hat{\mathcal{S}}')$ for $i \in [1, \ldots, k+1]$. From the first postcondition of Diff2 at line 10, we know that $\text{numHoles}(\hat{\mathcal{S}}') = |\hat{\Delta}|$. For any $i \in [1, \ldots, k]$, the first postcondition of Compose at line 12 implies $|\hat{\Delta}| = |\Delta'_i|$. Together, they imply that $\text{numHoles}(\hat{\mathcal{S}}') = |\Delta'_i|$.

The postcondition $\hat{\mathcal{S}}'[\Delta'_{k+1}] = \hat{\mathcal{S}}$ follows directly from the third postcondition of Diff2. Now consider Δ'_i for $i \in [1, \ldots, k]$. We know from the postcondition of Diff2 that $\text{numHoles}(\hat{\mathcal{S}}') = |\hat{\Delta}|$. Therefore, from the postcondition of Compose at line 18 (where we substitute $\hat{\mathcal{S}}'$ for the bound variable $\hat{\mathcal{S}}$), we know that $(\hat{\mathcal{S}}'[\hat{\Delta}])[\Delta_i] = \hat{\mathcal{S}}'[\Delta'_i]$. From the postconditon of Diff2 at line 10, we know $\hat{\mathcal{S}}'[\hat{\Delta}] = \hat{\mathcal{S}}$. Together, they imply $\hat{\mathcal{S}}[\Delta_i] = \hat{\mathcal{S}}'[\Delta'_i]$. □

Appendix E: Example of 4-way diff

We illustrate the 4-way *diff* using a simple example:

$$\begin{array}{lcl} O & \doteq & c\ ?\ \{x := 1\} : \{y := 2\}; z := 3 \\ A & \doteq & c\ ?\ \{x := 2\} : \{y := 2\}; \\ B & \doteq & c\ ?\ \{x := 1\} : \{y := 3\}; z := 3 \\ M & \doteq & c\ ?\ \{x := 2\} : \{y := 3\}; \end{array}$$

According to Algorithm 2, we start out with the shared program $\hat{\mathcal{S}} = O$ and $\Delta_O = [\,]$.

Now consider the first call to $\text{GenEdit}(\hat{\mathcal{S}}, A, \Delta_O)$. After invoking $\text{Diff2}(A, \hat{\mathcal{S}})$ at line 10, it returns the tuple $(\hat{\mathcal{S}}', \Delta, \hat{\Delta})$ where $\hat{\mathcal{S}}' \doteq c\ ?\ \{[\cdot]\} : \{y := 2\}; [\cdot]$, $\Delta \doteq [x := 2, \text{skip}]$ and $\hat{\Delta} \doteq [x := 1, z := 3]$.

The reader can verify that $\hat{\mathcal{S}}'[\Delta] = A$ and $\hat{\mathcal{S}}'[\hat{\Delta}] = \hat{\mathcal{S}} = O$. Next, consider the call to Compose$(\hat{\Delta}, \Delta_1)$ where $\Delta_1 = [\,]$. Therefore, the call to GenEdit returns the tuple $(\hat{\mathcal{S}}', [x := 1, z := 3], [x := 2, \text{skip}])$, which constitutes $\hat{\mathcal{S}}, \Delta_O, \Delta_{\mathcal{A}}$ for the next call to GenEdit.

The next call to GenEdit$(\hat{\mathcal{S}}, B, \Delta_O, \Delta_{\mathcal{A}})$ calls Diff2$(B, c\ ?\ \{[\cdot]\} : \{y := 2\}; [\cdot])$ and returns $(\hat{\mathcal{S}}', \Delta, \hat{\Delta})$, where $\hat{\mathcal{S}}' \doteq c\ ?\ \{[\cdot]\} : \{[\cdot]\}; [\cdot]$ and $\Delta \doteq [x := 1, y := 2, z := 3]$ (which becomes $\Delta_{\mathcal{B}}$) and $\hat{\Delta} \doteq [[\cdot], y := 2, [\cdot]]$. The reader can verify that $\hat{\mathcal{S}}'[\Delta] = B$ and $\hat{\mathcal{S}}'[\hat{\Delta}] = \hat{\mathcal{S}}$. The loop at line 11 updates Δ_O and $\Delta_{\mathcal{A}}$ — we only describe the latter. The return of Compose$(\hat{\Delta}, \Delta_{\mathcal{A}})$ updates $\Delta_{\mathcal{A}}$ to $[x := 2, y := 2, \text{skip}]$ by walking the first argument and replacing $[\cdot]$ with corresponding entry from $\Delta_{\mathcal{A}}$. Similarly, the Δ_O is updated by Compose$(\hat{\Delta}, \Delta_O)$ to $[x := 1, y := 2, z := 3]$.

The final call to GenEdit$(\hat{\mathcal{S}}, M, \Delta_O, \Delta_{\mathcal{A}}, \Delta_{\mathcal{B}})$ returns the tuple $(\hat{\mathcal{S}}, \Delta_O, \Delta_{\mathcal{A}}, \Delta_{\mathcal{B}}, \Delta_{\mathcal{M}})$, where $\hat{\mathcal{S}}, \Delta_O, \Delta_{\mathcal{A}}, \Delta_{\mathcal{B}}$ remain unchanged (since $\hat{\mathcal{S}}$ already contains holes at all the changed locations), and $\Delta_{\mathcal{M}}$ is assigned $[x := 2, y := 3, \text{skip}]$. The reader can verify that $\hat{\mathcal{S}}[\Delta_O] = O, \hat{\mathcal{S}}[\Delta_{\mathcal{A}}] = A, \hat{\mathcal{S}}[\Delta_{\mathcal{B}}] = B, \hat{\mathcal{S}}[\Delta_{\mathcal{M}}] = M$.

Appendix F: An abstract implementation of Diff2

Algorithm 3 Algorithm for 2-way AST Diff

```
1:  procedure Diff2(S, Ŝ)
2:      Input: A program S and a shared program Ŝ
3:      Output: Shared program Ŝ′ and edits Δ, Δ̂
4:      Ensures: |Δ| = |Δ̂| = numHoles(Ŝ′)
5:      Ensures: numHoles(Δ̂) = numHoles(Ŝ)
6:      Ensures: Ŝ′[Δ] = S, Ŝ′[Δ̂] = Ŝ
7:      if Ŝ = [·] then return ([·], [S], [Ŝ])
8:      else if Ŝ = S then return (S, [ ], [ ])
9:      else if * then return Diff2(S; skip, Ŝ)          ▷ Non-deterministic skip introduction
10:     else if * then return Diff2(skip; S, Ŝ)          ▷ Non-deterministic skip introduction
11:     else if * then return Diff2(S, Ŝ; skip)          ▷ Non-deterministic skip introduction
12:     else if * then return Diff2(S, skip; Ŝ)          ▷ Non-deterministic skip introduction
13:     else if S = S1; S2 and Ŝ = Ŝ1; Ŝ2 then
14:         (Ŝ1′, Δ1, Δ̂1) := Diff2(S1, Ŝ1)
15:         (Ŝ2′, Δ2, Δ̂2) := Diff2(S2, Ŝ2)
16:         return (Ŝ1′; Ŝ2′, Δ1 :: Δ2, Δ̂1 :: Δ̂2)
17:     else if S = C ? {S1} : {S2} and Ŝ = C′ ? {Ŝ1} : {Ŝ2} and C = C′ then
18:         (Ŝ1′, Δ1, Δ̂1) := Diff2(S1, Ŝ1)
19:         (Ŝ2′, Δ2, Δ̂2) := Diff2(S2, Ŝ2)
20:         return (C ? {Ŝ1′} : {Ŝ2′}, Δ1 :: Δ2, Δ̂1 :: Δ̂2)
21:     else if S = while(C) {S1} and Ŝ = while(C′) {Ŝ1} and C = C′ then
22:         (Ŝ1′, Δ1, Δ̂1) := Diff2(S1, Ŝ1)
23:         return (while(C) {Ŝ1′}, Δ1, Δ̂1)
24:     else
25:         return ([·], [S], [Ŝ])
```

Algorithm 3 describes Diff2 algorithm for computing the 2-way diff. It takes as input a program $\mathcal{S}$ and a program with holes $\hat{\mathcal{S}}$ and returns the shared program with holes $\hat{\mathcal{S}}'$ and edits Δ and $\hat{\Delta}$,

such that $\hat{\mathcal{S}}'[\Delta] = \mathcal{S}$ and $\hat{\mathcal{S}}'[\hat{\Delta}] = \hat{\mathcal{S}}$. Since $\hat{\mathcal{S}}$ may contain holes, the edit $\hat{\Delta}$ may contain holes. The algorithm recursively descends down the structure of the two programs and tries to identify the common program and generate respective edits for the differences. We use non-deterministic conditional to abstract from actual heuristics to match parts of the two ASTs. For example, when matching $\mathcal{S}$ with $\hat{\mathcal{S}}_1; \hat{\mathcal{S}}_2$, a heuristic may decide to match $\mathcal{S}$ with $\hat{\mathcal{S}}_1$ and create a shared program $[\cdot]; [\cdot]$ and edits $\Delta = [\mathcal{S}, \text{skip}]$, $\hat{\Delta} = [\hat{\mathcal{S}}_1, \hat{\mathcal{S}}_2]$; it may also choose to match $\mathcal{S}$ with $\hat{\mathcal{S}}_2$ and create a shared program $[\cdot]; [\cdot]$ and edits $\Delta = [\text{skip}, \mathcal{S}]$, $[\hat{\mathcal{S}}_1, \hat{\mathcal{S}}_2]$. The decision is often based on algorithms based on variants of *longest-common-subsequence* [Hirschberg 1977]. However, these decisions only help maximize the size of the shared program, and do not affect the soundness of the edit generation. Lines 9 to 12 allow us to model all such heuristics by non-deterministically inserting skip statemnets before or after a statement. Line 24 ensures that the diff procedure can always return by constructing the trivial shared program $[\cdot]$ and $\mathcal{S}$ and $\hat{\mathcal{S}}$ as the respective edits. Line 7 checks if $\hat{\mathcal{S}}$ is a hole, then the shared program is a hole $[\cdot]$ and the two edits contain $\mathcal{S}$ and $\hat{\mathcal{S}}$ respectively. Line 8 is the case when $\mathcal{S}$ equals $\hat{\mathcal{S}}$. We use = to denote the syntactic equality of the two syntax trees. The remaining rules are standard and recurse down the AST structure and match the subtrees.

11 ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers for their constructive feedback and useful suggestions. We would also like to thank Thomas Dillig and members of the UToPiA group for their comments on previous versions of this paper. We also gratefully acknowledge support from a Google Fellowship to the first author and from the National Science Foundation for supporting the second author on NSF Award CCF-1712067. The second author was also supported in part by AFRL Award #8750-14-2-0270. Finally, we would like thank Thomas Ball and James Larus for several discussions around the problem.

REFERENCES

Sven Apel, Olaf Lessenich, and Christian Lengauer. 2012. Structured Merge with Auto-tuning: Balancing Precision and Performance. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE 2012)*.

Sven Apel, Jörg Liebig, Benjamin Brandl, Christian Lengauer, and Christian Kästner. 2011. Semistructured Merge: Rethinking Merge in Revision Control Systems. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE '11)*.

Dimitar Asenov, Balz Guenat, Peter Müller, and Martin Otth. 2017. Precise version control of trees with line-based version control systems. In *International Conference on Fundamental Approaches to Software Engineering*. Springer, 152–169.

Gilles Barthe, Juan Manuel Crespo, and César Kunz. 2011. Relational verification using product programs. In *FM 2011: Formal Methods*. Springer, 200–214.

Gilles Barthe, Juan Manuel Crespo, and César Kunz. 2013. Beyond 2-safety: Asymmetric product programs for relational program verification. In *Logical Foundations of Computer Science*. Springer, 29–43.

Nick Benton. 2004. Simple relational correctness proofs for static analyses and program transformations. In *ACM SIGPLAN Notices*, Vol. 39. ACM, 14–25.

Yuriy Brun, Reid Holmes, Michael D Ernst, and David Notkin. 2013. Early detection of collaboration conflicts and risks. *IEEE Transactions on Software Engineering* 39, 10 (2013), 1358–1375.

Guilherme Cavalcanti, Paulo Borba, and Paola R. G. Accioly. 2017. Evaluating and improving semistructured merge. *PACMPL* 1, OOPSLA (2017), 59:1–59:27.

David Wheeler. 2017. The Apple goto fail vulnerability: lessons learned. http://www.dwheeler.com/essays/apple-goto-fail.html. (2017).

Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

Isil Dillig, Thomas Dillig, and Alex Aiken. 2011. Precise reasoning for programs using containers. In *ACM SIGPLAN Notices*, Vol. 46. ACM, 187–200.

H Christian Estler, Martin Nordio, Carlo A Furia, and Bertrand Meyer. 2013. Unifying configuration management with merge conflict detection and awareness systems. In *Software Engineering Conference (ASWEC), 2013 22nd Australian*. IEEE, 201–210.

Dennis Felsing, Sarah Grebing, Vladimir Klebanov, Philipp Rümmer, and Mattias Ulbrich. 2014. Automating regression verification. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*. 349–360.

Cormac Flanagan and K. Rustan M. Leino. 2001. Houdini, an Annotation Assistant for ESC/Java. In *FME 2001: Formal Methods for Increasing Software Productivity, International Symposium of Formal Methods Europe, Berlin, Germany, March 12-16, 2001, Proceedings*. 500–517.

Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. 2002. Extended Static Checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI '02)*. ACM, New York, NY, USA, 234–245. https://doi.org/10.1145/512529.512558

Fowler, Martin. 2011. Semantic Conflict. https://martinfowler.com/bliki/SemanticConflict.html. (2011).

Benny Godlin and Ofer Strichman. 2008. Inference rules for proving the equivalence of recursive procedures. *Acta Inf.* 45, 6 (2008), 403–439.

Mário Luís Guimarães and António Rito Silva. 2012. Improving early detection of software merge conflicts. In *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 342–352.

Chris Hawblitzel, Ming Kawaguchi, Shuvendu K. Lahiri, and Henrique Rebêlo. 2013. Towards Modularly Comparing Programs Using Automated Theorem Provers. In *Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings (Lecture Notes in Computer Science)*, Vol. 7898. Springer, 282–299.

Daniel S. Hirschberg. 1977. Algorithms for the Longest Common Subsequence Problem. *J. ACM* 24, 4 (1977), 664–675.

Susan Horwitz, Jan Prins, and Thomas Reps. 1989. Integrating noninterfering versions of programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 11, 3 (1989), 345–387.

Daniel Jackson and David A. Ladd. 1994. Semantic Diff: A Tool for Summarizing the Effects of Modifications. In *Proceedings of the International Conference on Software Maintenance, ICSM 1994, Victoria, BC, Canada, September 1994*. IEEE Computer Society, 243–252.

John Gruber. 2014. On the Timing of iOS's SSL Vulnerability. https://daringfireball.net/2014/02/apple_prism. (2014).

Sanjeev Khanna, Keshav Kunal, and Benjamin C Pierce. 2007. A formal investigation of diff3. In *FSTTCS 2007: Foundations of Software Technology and Theoretical Computer Science*. Springer, 485–496.

Knoy, Gabriel. 2012. How Often Does Gitmerge make mistakes? https://news.ycombinator.com/item?id=9871042. (2012).

Shuvendu K. Lahiri, Chris Hawblitzel, Ming Kawaguchi, and Henrique Rebêlo. 2012. SYMDIFF: A Language-agnostic Semantic Diff Tool for Imperative Programs. In *Proceedings of the 24th International Conference on Computer Aided Verification (CAV'12)*.

Shuvendu K. Lahiri, Kenneth L. McMillan, Rahul Sharma, and Chris Hawblitzel. 2013. Differential assertion checking. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*. ACM, 345–355.

Olaf Lebetaenich, Sven Apel, and Christian Lengauer. 2015. Balancing Precision and Performance in Structured Merge. *Automated Software Engg.* 22, 3 (Sept. 2015).

Lee, TK. 2012. The Problem of Automatic Code Merging. http://www.personal.psu.edu/txl20/blogs/tks_tech_notes/2012/03/the-problem-of-automatic-code-merging.html. (2012).

Lenski, Dan. 2015. Is it possible for Git merging to make a mistake without detecting a conflict? https://www.quora.com/Is-it-possible-for-Git-merging-to-make-a-mistake-without-detecting-a-conflict. (2015).

Francesco Logozzo, Shuvendu K. Lahiri, Manuel Fähndrich, and Sam Blackshear. 2014. Verification modulo versions: towards usable verification. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*. ACM, 32.

Lutton, Mark. 2014. Infinite Loop Caused by Git Merge. https://stackoverflow.com/questions/23523713/how-can-i-trust-git-merge. (2014).

Ahmed-Nacer Mehdi, Pascal Urso, and François Charoy. 2014. Evaluating software merge quality. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*. ACM, 9.

Nimrod Partush and Eran Yahav. 2014. Abstract semantic differencing via speculative correlation. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*. ACM, 811–828.

Suzette Person, Matthew B. Dwyer, Sebastian G. Elbaum, and Corina S. Pasareanu. 2008. Differential symbolic execution. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2008, Atlanta, Georgia, USA, November 9-14, 2008*. ACM, 226–237.

David A. Ramos and Dawson R. Engler. 2011. Practical, Low-Effort Equivalence Verification of Real Code. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*. 669–685.

Reddit. 2017a. Automatic Merge Mistakes. https://www.reddit.com/r/git/comments/5bssjv/automatic_merge_mistakes/. (2017).

Reddit. 2017b. How Do you Deal with Auto Merge? https://www.reddit.com/r/git/comments/5hn80k/how_do_you_deal_with_auto_merge/. (2017).

Rostedt, Steven. 2011. Fix Bug Caused by Git Merge. http://lkml.iu.edu/hypermail/linux/kernel/1106.0/00645.html. (2011).

SlashDot. 2014. Apple SSL Bug In iOS Also Affects OS X. http://apple.slashdot.org/story/14/02/22/2143224/apple-ssl-bug-in-ios-also-affects-os-x. (2014).

Marcelo Sousa and Isil Dillig. 2016. Cartesian Hoare logic for verifying k-safety properties. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 57–69.

Tachio Terauchi and Alex Aiken. 2005. *Secure information flow as a safety problem*. Springer.

Tim Wood, Sophia Drossopoulou, Shuvendu K. Lahiri, and Susan Eisenbach. 2017. Modular Verification of Procedure Equivalence in the Presence of Memory Allocation. In *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*. 937–963.

Hongseok Yang. 2007. Relational separation logic. *Theoretical Computer Science* 375, 1 (2007), 308–334.

Wuu Yang, Susan Horwitz, and Thomas Reps. 1990. A Program Integration Algorithm That Accommodates Semantics-preserving Transformations. *SIGSOFT Softw. Eng. Notes* 15, 6 (Oct. 1990), 133–143.

Anna Zaks and Amir Pnueli. 2008. Covac: Compiler validation by program analysis of the cross-product. In *FM 2008: Formal Methods*. Springer, 35–51.