A Mechanically Checked Generation of Correlating Programs Directed by Structured Syntactic Differences

Thibaut Girka, David Mentré, Yann Régis-Gianas

A Mechanically Checked Generation of Correlating Programs Directed by Structured Syntactic Differences

Thibaut Girka[1,2], David Mentré[1], and Yann Régis-Gianas[2]

[1]Mitsubishi Electric R&D Centre Europe, F-35708 Rennes, France
[2]Univ Paris Diderot, Sorbonne Paris Cité, PPS, UMR 7126 CNRS, PiR2, INRIA Paris-Rocquencourt, F-75205 Paris, France

Abstract. We present a new algorithm for the construction of a correlating program from the syntactic difference between the original and modified versions of a program. This correlating program exhibits the semantics of the two input programs and can then be used to compute their semantic differences, following an approach of Partush and Yahav [12]. We show that Partush and Yahav's correlating program is unsound on loops that include an early exit. Our algorithm is defined on an imperative language with **while**-loops, **break**, and **continue**. To guarantee its correctness, it is formalized and mechanically checked within the Coq proof assistant. On a series of examples, we experimentally find that the static analyzer `dizy` is at least as precise on our correlating program as on Partush and Yahav's.

1 Introduction

Most of current software engineering practices are using textual differences – as provided by the Unix `diff` tool – to examine changes made on a program. However, many development tasks [9] would take benefit from *semantic differences*, i.e. a representation of the *meaning* of each change made on a program. Such a piece of information would be very helpful to check that refactoring does not change current program behavior [15], to minimize replay of tests after a change [1], to check that a change indeed modified the expected parts of the program [7,8], to generate synthetic patches for code reviews, to determine security impact of a change [2], etc.

Moreover, the tool computing and presenting such semantic differences to a final user should be *sound*, i.e. no change should be missed, so that the user can safely elaborate further software engineering tasks on top of the reported differences. Building such a sound tool implies that the underlying theory is itself sound and that it is correctly used to specify and implement the tool. This is an error-prone task which can be nonetheless guided by modern proof assistants. This approach leads to Correct-by-Construction tools like the CompCert C compiler [10].

We follow Partush's and Yahav [12]'s approach to compute semantic differences between two variants P and P' of the same program: (i) we generate a

correlating program C that tightly interleaves the instructions to simulate the parallel executions of P and P'; (ii) we apply a dedicated Abstract Interpretation [3] technique on C to approximate the correlation between the variables of P and P'.

Even though we reuse an existing approach, we propose a new algorithm to build the correlating program and we formally verified this algorithm within the `Coq` proof assistant [11]. Incidentally, this allowed us to find that Partush and Yahav's approach produces incorrect results for loop constructions with early exit. In our opinion, Parush and Yahav's approach is flawed by design because their generation algorithm is directed by a *textual* difference between the two versions of the program. This textual representation is not structured enough to correctly perform a static scheduling of the two programs instructions. On the contrary, our generation algorithm is directed by a *structured* difference between abstract syntax trees which is a better input for a static scheduling because it gives a structured relationship between the control flow graphs of the two programs. Besides, our tool is extracted from our `Coq` formalization and is thus Correct-by-Construction. To sum up, the contributions of our paper are as follow: (i) we identify errors in previous work of Partush and Yahav [12]; (ii) we propose a new sound algorithm for the production of a correlating program from a syntactic structural difference; (iii) both the underlying theory and our algorithm are formalized and mechanically checked within the Coq proof assistant; (iv) we experimentally compare the quality of our approach against Partush and Yahav's work.

Plan In Section 2, we give an overview of our analysis chain structure along with an example. In Section 3, we present our input language and its guarded form as well as their semantics. In Section 4, we show how we represent syntactic differences between the original and modified programs. In Section 5, we present the core of our approach to generate a sound correlating program. In Section 6, we discuss our implementation and some experiments. In Section 7, we present related work and finally present our conclusion and future work in Section 8.

2 Overview

The original analysis described by Nimrod Partush and Eran Yahav [12] can be decomposed in two main components: a set of program transformations (implemented in a tool called `ccc`) yielding a *correlating program*, and a static analysis on this correlating program (implemented in a tool called `dizy`). Our work mainly consists in a restricted (but proven correct) replacement for the first part (`ccc`) while producing an output that remains compatible with the second part (`dizy`). This new algorithm called `correlating_program` uses structured syntactic differences to correctly interleave the instructions of the two programs while factoring their control flow as much as possible. In this section, we compare Partush's and Yahav's algorithm and our algorithm on the following examples where the line "`a = a + 10;`" is moved.

Original version

```
void f(int a, int b) {
  if (a < b) {
    a = a + 10; // moved line
    b = b - 10;
  } else {
    b = b + 1;
  };
}
```

Modified version

```
void f(int a, int b) {
  if (a < b) {
    b = b - 10;
  } else {
    a = a + 10; // moved line
    b = b + 1;
  };
}
```

Textual diff-directed tool `ccc` First, `ccc` turns the two programs into a semantically equivalent guarded form where every assignment is guarded by a condition. Thus, conditional statements are useless and the only remaining control structures are loops. On the example, this step produces the following two programs where the `if (a < b)` statement has been translated as a guard for each statement:

Guarded original program (ccc)

```
void f(int a, int b) {

  Guard G0 = (a < b);
  if (G0) a = a + 10;
  if (G0) b = b - 10;
  if (!G0) b = b + 1;
}
```

Guarded modified program (ccc)

```
void f(int a, int b) {
  int T_a = a; int T_b = b;
  Guard T_G0 = (T_a < T_b);
  if (T_G0) T_b = T_b - 10;
  if (!T_G0) T_a = T_a + 10;
  if (!T_G0) T_b = T_b + 1;
}
```

Second, `ccc` interleaves the instructions of these two guarded programs according to their *textual difference* into the following correlating program.

Correlating program (ccc)

```
void f(int a, int b) {
  int T_a = a; int T_b = b;
  Guard G0 = (a < b);          // if condition (original version)
  Guard T_G0 = (T_a < T_b);    // if condition (modified version)
  if (G0) a = a + 10;          // then branch (original version)
  if (G0) b = b - 10;          // then branch (original version)
  if (T_G0) T_b = T_b - 10;    // then branch (modified version)
  if (!T_G0) T_a = T_a + 10;   // else branch (modified version)
  if (!G0) b = b + 1;          // else branch (original version)
  if (!T_G0) T_b = T_b + 1;    // else branch (modified version)
}
```

Syntactic difference-directed tool `correlating_program` Instead of a textual difference, our algorithm takes a *syntactic difference between the abstract syntax trees* of two programs as defined in Section 4. First, it *tags* variables in this syntax difference to avoid naming conflicts. Tagging consists in prefixing names of the original program with `O_` and names of the modified program with `T_O_`:

Structured difference

```
 void f(int a, int b) {
   if (a < b) {
-    a = (a + 10);
     b = (b - 10);
   } else {
+    a = (a + 10);
     b = (b + 1);
   };
 }
```

Tagged structured difference

```
 void test(int O_a, int O_b) {
   if ([O_a < O_b] → [T_O_a < T_O_b]) {
-    O_a = (O_a + 10);
     O_b = O_b - 10 → T_O_b = T_O_b - 10;
   } else {
+    T_O_a = (T_O_a + 10);
     O_b = O_b + 1 → T_O_b = T_O_b + 1;
   };
 }
```

The correlating program is then obtained by recursively translating to guarded form the original and modified programs represented by the structured difference, and interlacing them following the difference structure, as defined in Section 5, leading in this case to the same correlating program as produced by `ccc`.

Motivating example The correctness proof of the above program transformations ensures that the execution of the correlating program correctly simulates the input programs parallel execution. Such transformations are error-prone, and it is easy to overlook corner cases, which is what happens in `ccc` regarding early exits of loops. Indeed, consider the following example, in which the user simply adds a **break** statement so that the loop is iterated at most once in the modified version of the program:

Original version

```
void fail(int x) {
  i = 0;
  while (i <= 1) {
    i = i + 1;
    x = x + 1;

  }
}
```

Modified version

```
void fail(int x) {
  i = 0;
  while (i <= 1) {
    i = i + 1;
    x = x + 1;
    break; // added break statement
  }
}
```

In this case, we observe that the following correlating program produced by `ccc` will return as soon as the code corresponding to the modified version reaches the **break** statement (translated to a **goto** statement on line 15), thus incorrectly simulating the original version of the program. Meanwhile, our tool `correlating_program` produces a sound correlating program by translating the **break** statement into a guard affectation (`T_G0 = 0` on line 15) inhibiting further execution of the code corresponding to the modified version of the loop without impacting the execution of the code corresponding to the original version of the loop:

ccc

```
1  void fail(int x) {
2   int T_x = x;
3   int i = 0; int T_i = 0;
4
5
6  L1: T_L1:;
7   Guard G0 = (i <= 1);
8   Guard T_G0 = (T_i <= 1);
9
10
11  if (G0) i = i + 1;
12  if (T_G0) T_i = T_i + 1;
13  if (G0) x = x + 1;
14  if (T_G0) T_x = T_x + 1;
15  if (T_G0) goto T_L3;
16
17
18  if (G0) goto L1;
19  if (T_G0) goto T_L1;
20 L3: T_L3:
21 }
```

correlating_program

```
void fail(int O_x) {
 int T_O_x = O_x;
 int O_i = 0; int T_O_i = 0;

 Guard G1 = 1; Guard T_G1 = 1;
 Guard G0 = (O_i <= 1);
 Guard T_G0 = (T_O_i <= 1);
L1:
 if (G0) G1 = 1; // unused here (used to
 if (T_G0) T_G1 = 1; // encode continue)
 if (G0) if (G1) O_i = O_i + 1;
 if (T_G0) if (T_G1) T_O_i = T_O_i + 1;
 if (G0) if (G1) O_x = O_x + 1;
 if (T_G0) if (T_G1) T_O_x = T_O_x + 1;
 if (T_G0) if (T_G1) T_G0 = 0;
 if (G0) G0 = (O_i <= 1);
 if (T_G0) T_G0 = (T_O_i <= 1);
 if (G0) goto L1;
 if (T_G0) goto L1;

}
```

3 Source language and its guarded form

In this section, we define the syntax and semantics of our input language as well as its guarded form, that will later be used in our translation functions.

3.1 Input language

Syntax We write x for a variable identifier taken in an enumerable set of identifiers $\mathcal{I}$, $\mathbf{n}$ for an integer value taken in $\mathbb{Z}$ and $\mathbf{b}$ for a Boolean value in $\{\textsf{true}, \textsf{false}\}$. As shown in Figure 1, the input language is a standard imperative programming language with **while**-loops and **if**-statements which also includes the non-local control-flow operators **break** and **continue**.

Semantics The input language enjoys a standard big-step semantics: a program transforms the store by means of commands and commands make use of pure expressions to perform arithmetic and boolean computations. A store $S : \mathcal{I} \to \mathbb{Z}$ is a partial map from variable identifiers to integer values. The empty store is written $\emptyset$, $\forall x \in \mathrm{dom}(S)$ quantifies over the finite domain of S, and $S[x \mapsto v]$ is the store defined on $\mathrm{dom}(S) \cup \{x\}$ such that $S[x \mapsto v](y) = v$ if $x = y$ and $S(y)$ otherwise. The judgment "$S \vdash e \Downarrow \mathbf{n}$" is read "In the store S, the arithmetic expression e evaluates into the integer $\mathbf{n}$." and the judgment "$S \vdash b \Downarrow \mathbf{b}$" is read "In the store S, the boolean expression b evaluates into the boolean $\mathbf{b}$." The interpretation of a command yields a return *mode* m which is either *normal* (written $\square$), *interrupted* (written $\star$, used to handle **break** statements) or *continuing* (written $\circ$, used to handle **continue** statements). The judgment "$S_0 \vdash c \Downarrow_m S_1$" is read "The command c transforms S_0 into S_1 in mode m".

3.2 Guarded language

We derive a guarded language from the input language of the previous section. Every condition is now stored into a boolean variable called a *guard variable* and every atomic instruction is guarded by a conjunction of guard variables (called a *guard* in the sequel). This specific form effectively abstracts execution paths into guard variables, as the values of the guard variables precisely determine a single block in the control flow graph of the input program. Thus, assigning specific values to these guard variables activates specific instructions of the input program. In Section 5, this mechanism will be at the heart of the static interleaving of programs instructions.

Syntax To simplify the proof of some technical lemmas, the identifiers of guard variables are taken in the set $\mathcal{I}_\mathcal{G}$ of words over the alphabet $\{0, 1\}$. We will use the fact that a word in this alphabet can encode a path in an abstract syntax tree. We write g for such guard identifiers.

Definition 1 (Guard identifier independance). *A guard g is independant for a path π, written $g\#\pi$, if it does not end with π.*

Syntax

$$c ::= a \mid c; c \mid \textsf{while}\ (b)\ c \mid \textsf{if}\ (b)\ c\ \textsf{else}\ c \qquad \text{(Commands)}$$
$$a ::= \textsf{skip} \mid x = e \mid \textsf{break} \mid \textsf{continue} \qquad \text{(Atomic commands)}$$
$$b ::= \textsf{true} \mid \textsf{false} \mid b \,\&\&\, b \mid !b \mid e \leq e \qquad \text{(Boolean Expressions)}$$
$$e ::= x \mid \mathbf{n} \mid e + e \qquad \text{(Arithmetic Expressions)}$$

Big-step semantics for arithmetic expressions

$$\textsc{Cst}\ \frac{}{S \vdash \mathbf{n} \Downarrow \mathbf{n}} \qquad \textsc{Var}\ \frac{}{S \vdash x \Downarrow S(x)} \qquad \textsc{Sum}\ \frac{S \vdash e_1 \Downarrow \mathbf{n}_1 \qquad S \vdash e_2 \Downarrow \mathbf{n}_2}{S \vdash e_1 + e_2 \Downarrow \mathbf{n}_1 +_{\mathbb{Z}} \mathbf{n}_2}$$

Big-step semantics for boolean expressions

$$\textsc{Cst}\ \frac{}{S \vdash \mathbf{b} \Downarrow \mathbf{b}} \qquad \textsc{Not}\ \frac{S \vdash b \Downarrow \mathbf{b}}{S \vdash !b \Downarrow \neg \mathbf{b}} \qquad \textsc{And}\ \frac{S \vdash b_1 \Downarrow \mathbf{b}_1 \quad S \vdash b_2 \Downarrow \mathbf{b}_2}{S \vdash b_1 \,\&\&\, b_2 \Downarrow \mathbf{b}_1 \wedge \mathbf{b}_2} \qquad \textsc{LessEqual}\ \frac{S \vdash e_1 \Downarrow \mathbf{n}_1 \quad S \vdash e_2 \Downarrow \mathbf{n}_2}{S \vdash e_1 \leq e_2 \Downarrow \mathbf{n}_1 \leq_{\mathbb{Z}} \mathbf{n}_2}$$

Big-step semantics for commands

$$\textsc{Skip}\ \frac{}{S \vdash \textsf{skip} \Downarrow_\square S} \qquad \textsc{Assign}\ \frac{S \vdash e \Downarrow \mathbf{n}}{S \vdash x = e \Downarrow_\square S[x \mapsto \mathbf{n}]}$$

$$\textsc{Seq}\ \frac{S \vdash c_1 \Downarrow_\square S' \qquad S' \vdash c_2 \Downarrow_m S''}{S \vdash c_1; c_2 \Downarrow_m S''} \qquad \textsc{Seq Interrupted}\ \frac{S \vdash c_1 \Downarrow_m S'}{S \vdash c_1; c_2 \Downarrow_m S'}\ m \neq \square$$

$$\textsc{Then}\ \frac{S \vdash c_1 \Downarrow_m S' \qquad S \vdash b \Downarrow \textsf{true}}{S \vdash \textsf{if}\ (b)\ c_1\ \textsf{else}\ c_2 \Downarrow_m S'} \qquad \textsc{Else}\ \frac{S \vdash c_2 \Downarrow_m S' \qquad S \vdash b \Downarrow \textsf{false}}{S \vdash \textsf{if}\ (b)\ c_1\ \textsf{else}\ c_2 \Downarrow_m S'}$$

$$\textsc{While False}\ \frac{S \vdash b \Downarrow \textsf{false}}{S \vdash \textsf{while}\ (b)\ c \Downarrow_\square S} \qquad \textsc{While True}\ \frac{S \vdash b \Downarrow \textsf{true} \quad S \vdash c \Downarrow_m S' \qquad S' \vdash \textsf{while}\ (b)\ c \Downarrow_\square S''}{S \vdash \textsf{while}\ (b)\ c \Downarrow_\square S''}\ m \neq \star$$

$$\textsc{While Break}\ \frac{S \vdash b \Downarrow \textsf{true} \qquad S \vdash c \Downarrow_\star S'}{S \vdash \textsf{while}\ (b)\ c \Downarrow_\square S'} \qquad \textsc{Break}\ \frac{}{S \vdash \textsf{break} \Downarrow_\star S} \qquad \textsc{Continue}\ \frac{}{S \vdash \textsf{continue} \Downarrow_\circ S}$$

Fig. 1. Syntax and semantics of the input language.

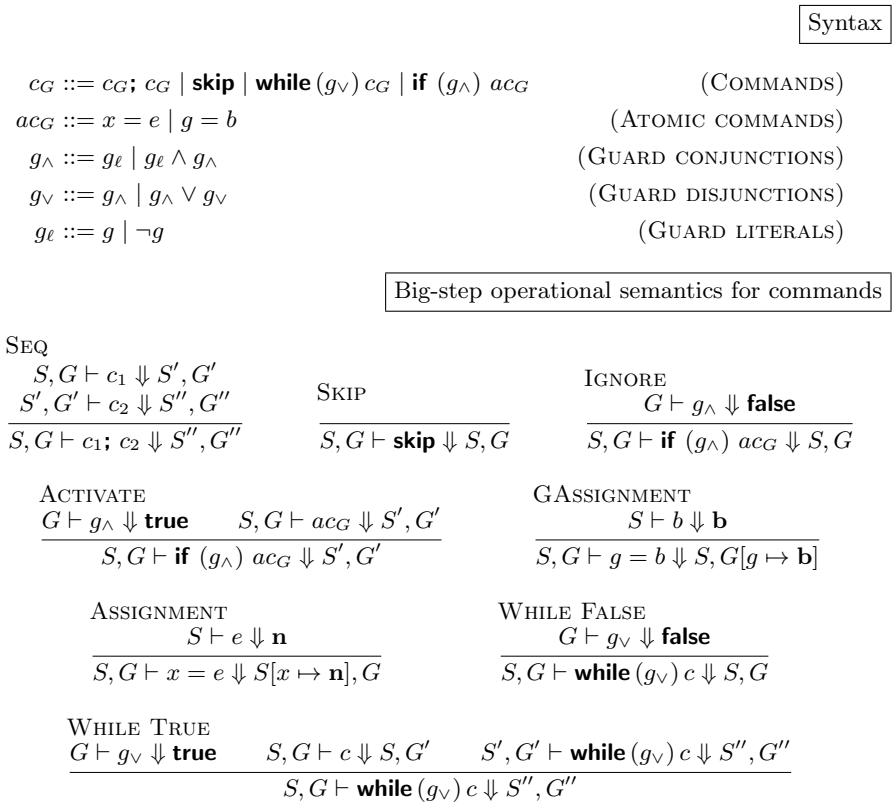

Fig. 2. Syntax and semantics of the guarded language.

The syntax of the guarded language includes an assignment statement ac_G guarded by a conjunction of (positive or negative) guard variables and a **while**-loop statement guarded by a disjunction of guard conjunctions. Notice that **break** and **continue** are not present in the guarded language.

Semantics Besides the store S of integer variables, a program in the guarded language also transforms a store G of guarded variables, which is a partial map from guard identifiers to boolean values. The operations over standard memories are naturally transported to guard memories. The judgment "$S_0, G_0 \vdash c_G \Downarrow S_1, G_1$" is read "The guarded command c_G transforms the store S_0 and the guard store G_0 into a new store S_1 and a new guard store G_1". The rules for the evaluation of guards and disjunctions of guards are straightforward and thus omitted.

3.3 Translation to guarded form

We transform input programs into guarded form using a recursive translation function CI defined as follows:

c	o	$CI(gl,\ \pi,\ c,\ o)$
skip	o	**skip**
$x = e$	o	**if** (gl) $x = e$
c_1**;** c_2	o	$CI(gl,\ 0 \cdot \pi,\ c_1,\ o)$**;** $CI(gl,\ 1 \cdot \pi,\ c_1,\ o)$
if (b) c_1 **else** c_2	o	**if** (gl) $\pi = b$**;** $CI(gl \wedge \pi,\ 1 \cdot \pi,\ c_1,\ o)$**;** $CI(gl \wedge \neg\pi,\ 0 \cdot \pi,\ c_2,\ o)$
while (b) c	o	**if** (gl) $\pi = b$**;** **while** $(gl \wedge \pi)$ { **if** $(gl \wedge \pi)$ $1 \cdot \pi =$ **true;** $CI(gl \wedge \pi \wedge (1 \cdot \pi),\ 1 \cdot 1 \cdot \pi,\ c,$ **Some** $\pi)$**;** **if** $(gl \wedge \pi)$ $\pi = b$ }
break	**Some** π'	**if** (gl) $\pi' =$ **false**
continue	**Some** π'	**if** (gl) $1 \cdot \pi' =$ **false**

$CI(gl,\ \pi,\ c,\ o)$ is a guarded program simulating the input program c ; π is the path of the sub-program c in the whole program and is used as a fresh name for new guard variables ; gl is the guard conjunction guarding c in the program ; o equals "**Some** π'" if the innermost loop under which c is executed is at path π' or "**None**" if there is no such loop. o is used in the translation of **break** and **continue** statements by keeping track of the guard variables π' and $1 \cdot \pi'$ controlling the execution of the innermost loop.

Let us consider the case where c is "**if** (b) c_1 **else** c_2". We first create a new guard of name π to which we conditionally assign the value b under the guard conjunction gl: "**if** (gl) $\pi = b$". This guard represents the condition used to select the branch of the **if** statement. Then we recursively call CI on c_1 (the **then** part), guarded by the conjunction of the previous guards and the newly created guard ("$gl \wedge \pi$"). We create a new unique path "$1 \cdot \pi$" for the program translated under this path. We do the same for c_2 (the **else** part), negating the guard π ("$gl \wedge \neg\pi$") and creating another unique path "$0 \cdot \pi$".

The soundness of CI is expressed by the following technical Lemma. Roughly speaking, this lemma states that $CI(gl,\ \pi,\ c,\ o)$ simulates c correctly assuming that the guard gl is active (i). The assumptions (ii) and (iii) are extra invariants that makes the transformation work: o must provide the guard for the innermost loop (ii) and the guard identifiers in gl must be independent with respect to the path π. The extra conclusions (a),(b) and (c) ensure that the invariants are maintained by each evaluation step of the guarded program.

Lemma 1 (CI is sound on active guards). *Coq: CP.CI_sound*
If $S \vdash c \Downarrow_m S'$ *holds; (i)* $G \vdash gl \Downarrow$ **true** *holds; (ii)* $m \neq \square \implies \exists \pi_l, o =$ **Some** π_l *; (iii)* $\forall\, \pi_l,\ o =$ **Some** $\pi_l \implies G(\pi_l) = G(1 \cdot \pi_l) =$ **true** *; and (iv)* $\forall g \in gl, g\#\pi$ *; then there exists a guard store* G' *such that* $S, G \vdash CI(gl,\ \pi,\ c,\ o) \Downarrow S', G'$ *holds and (a)* $\forall g \in dom(G), g\#\pi \implies G'(g) = G(g)$ *; (b)* $m = \star \implies G'(\pi_l) =$ **false** *; and (c)* $m = \circ \implies G'(1 \cdot \pi_l) =$ **false**.

4 Structured differences between programs

The syntactic differences between the abstract syntax trees of two syntactically correct programs is denoted by a special representation of a whole program together with a patch. This representation will be processed in a purely recursive way by the algorithm that generates the correlating program.

Syntax The structured difference language is derived from the input language, in such a way that each internal node of the abstract syntax tree is associated with a local mutation Δ (where $\pm ::= - \mid +$):

$$\begin{aligned}\Delta ::= {} & \pm[c]; \Delta \mid \pm\Delta; [c] \mid \Delta; \Delta \mid a \to a' \\ & \mid \textbf{if } (b \to b') \ \Delta \ \textbf{else } \Delta \mid \textbf{while } (b \to b') \ \Delta \\ & \mid \pm[\textbf{if } (b) \ c \ \textbf{else}] \ \Delta \mid \pm[\textbf{if } (b)] \ \Delta \ [\textbf{else } c] \mid \pm[\textbf{while } (b)] \ \Delta\end{aligned}$$

The notation "$\pm[c]; \Delta$" means that the command c is removed from the original program ("$-[c]; \Delta$") or inserted into the modified program ("$+[c]; \Delta$") while the right side is kept with a local mutation Δ. The notation "$a \to a'$" means that the leaf command a of the original program is replaced by "a'". "$+[\textbf{if } (b) \ c \ \textbf{else}] \ \Delta$" means that an **if** statement is inserted in the modified program with the command c as its **then** branch and using the existing code (with a local mutation) Δ as its **else** branch.

Semantics A structured difference represents the full original program along with the differences leading to the modified program. The projection function Π_0 (resp. Π_1) returns the original (resp. modified) embedded program:

Δ	$\Pi_0(\Delta)$	$\Pi_1(\Delta)$
$-[c]; c'$	$c; \Pi_0(c')$	$\Pi_1(c')$
$-c; [c']$	$\Pi_0(c); c'$	$\Pi_1(c)$
$+[c]; c'$	$\Pi_0(c')$	$c; \Pi_1(c')$
$+c; [c']$	$\Pi_0(c)$	$\Pi_1(c); c'$
$c; c'$	$\Pi_0(c); \Pi_0(c')$	$\Pi_1(c); \Pi_1(c')$
if $(b \to b')$ c **else** c'	**if** (b) $\Pi_0(c)$ **else** $\Pi_0(c')$	**if** (b') $\Pi_1(c)$ **else** $\Pi_1(c')$
while $(b \to b')$ c	**while** (b) $\Pi_0(c)$	**while** (b') $\Pi_1(c)$
$+[\textbf{if } (b) \ c \ \textbf{else}] \ c'$	$\Pi_0(c')$	**if** (b) c **else** $\Pi_1(c')$
$+[\textbf{if } (b)] \ c \ [\textbf{else } c']$	$\Pi_0(c)$	**if** (b) $\Pi_1(c)$ **else** c'
$-[\textbf{if } (b) \ c \ \textbf{else}] \ c'$	**if** (b) c **else** $\Pi_0(c')$	$\Pi_1(c')$
$-[\textbf{if } (b)] \ c \ [\textbf{else } c']$	**if** (b) $\Pi_0(c)$ **else** c'	$\Pi_1(c)$
$+[\textbf{while } (b)] \ c$	$\Pi_0(c)$	**while** (b) $\Pi_1(c)$
$-[\textbf{while } (b)] \ c$	**while** (b) $\Pi_0(c)$	$\Pi_1(c)$
$a \to a'$	a	a'

A difference between two programs can always be found. Section 6 explains how we tackled this problem.

Theorem 1 (Completeness of the diff. language). *Coq: CP.diff_completeness*
For all pair of programs (p, p'), there exists a difference Δ such that $\Pi_0(\Delta) = p$ and $\Pi_1(\Delta) = p'$.

5 Generation algorithm directed by structured differences

We define in Figure 3 and Figure 4 a correlating program generation algorithm directed by structured differences as a recursive function CP. The program $CP(\Delta, \pi_0, \pi_1, gl_0, gl_1, o_0, o_1)$ is a correlating program of $\Pi_0(\Delta)$ and $\Pi_1(\Delta)$, corresponding to an interleaving of the guarded forms $CI(gl_0, \pi_0, \Pi_0(\Delta), o_0)$ and $CI(gl_1, \pi_1, \Pi_1(\Delta), o_1)$. π_0 and π_1 are the paths of the subprograms $\Pi_0(\Delta)$ and $\Pi_1(\Delta)$ in the whole correlating program. They are used as fresh names for new guard variables, which also eases Coq proofs by encoding path information in the name of the guards. gl_0 and gl_1 are the guard conjunctions guarding $\Pi_0(\Delta)$ and $\Pi_1(\Delta)$ in the whole correlating program, while o_0 and o_1 represent the optional innermost loop under which $\Pi_0(\Delta)$ and $\Pi_1(\Delta)$ are executed, and are used to translate **break** and **continue** statements.

For example, consider the definition of "$-$[**if** (b) c **else**] Δ" in Figure 3, corresponding to the removal of an **if** statement and its **then** part while keeping its **else** part. We first create a new guard π_0 to which we conditionally assign the value b under the conjunction gl_0 of the original program ("**if** (gl_0) $\pi_0 = b$") because the **if** statement is only executed in the first program. We then call CI to output the guarded form of the statement c under the removed **then** part: "$CI(gl_0 \wedge \pi_0, 1 \cdot \pi_0, c, o_0)$". It will be conditionally executed under the conjunction of gl_0 and the new guard π_0, under a new unique path "$1 \cdot \pi_0$". We then continue the translation of the remaining structured difference Δ by recursively calling CP on it: "$CP(\Delta, 0 \cdot \pi_0, \pi_1, gl_0 \wedge \neg\pi_0, gl_1, o_0, o_1)$". For the original program, we create a new unique path "$0 \cdot \pi_0$". This part is guarded by "$gl_0 \wedge \neg\pi_0$" as it is executed under the **else** part of the original program. For the modified program, we keep the guard gl_1 and the path π_1 which is still unique. o_0 and o_1 are reused unmodified as we are not translating a **while** statement.

Definition 2 (Store splitting). *Two stores S_0 and S_1 are said to split a store S if $\forall\, n \in \{0,1\},\ \forall x \in dom(S_n),\ S(x) = S_n(x)$; and S_0 contains only variables starting with "`O_`", and S_1 with "`T_O_`".*

Definition 3 (Guarded store splitting). *Two guarded stores G_0 and G_1 are said to split a guarded store G if $\forall\, n \in \{0,1\},\ \forall x \in dom(G_n),\ G(x) = G_n(x)$; and G_0 contains only variables ending with 0, and G_1 with 1.*

Lemma 2 (CP is sound under context). `Coq: CP.cp_sound`
For all S_0, S_0', S_1, S_1', S, S', G_0, G_0', G_1, G_1', G, G', π_0, π_1, gl_0, gl_1, o_0, o_1, if (i) $\forall\, n \in \{0,1\},\ S_n, G_n \vdash CI(gl_n, \pi_n, \Pi_n(\Delta), o_n) \Downarrow S_n', G_n'$ holds ; (ii) S_0 and S_1 split S ; (iii) G_0 and G_1 split G ; (iv) Variables appearing in $\Pi_0(\Delta)$ start with "`O_`", those of $\Pi_1(\Delta)$ with "`T_O_`" ; (v) gl_0 contains only variables ending with 0, and gl_1 with 1 ; then there exist a store S' and a guard store G' such that (a) $S, G \vdash CP(\Delta, \pi_0, \pi_1, gl_0, gl_1, o_0, o_1) \Downarrow S', G'$ holds ; (b) S_0' and S_1' split S' ; (c) G_0' and G_1' split G'.

While the above key lemma mentions the invariants ((ii), (iii), (iv), (v), (b) and (c)) used in the induction, CP is typically used with fixed initial values for most of its arguments, hence the following theorem.

Δ	$CP(\Delta,\ \pi_0,\ \pi_1,\ gl_0,\ gl_1,\ o_0,\ o_1)$
$-[c];\Delta$	$CI(gl_0,\ 0\cdot\pi_0,\ c,\ o_0)$; $CP(\Delta,\ 1\cdot\pi_0,\ \pi_1,\ gl_0,\ gl_1,\ o_0,\ o_1)$
$-\Delta;[c]$	$CP(\Delta,\ 0\cdot\pi_0,\ \pi_1,\ gl_0,\ gl_1,\ o_0,\ o_1)$; $CI(gl_0,\ 1\cdot\pi_0,\ c,\ o_0)$
$+[c];\Delta$	$CI(gl_1,\ 0\cdot\pi_1,\ c,\ o_1)$; $CP(\Delta,\ \pi_0,\ 1\cdot\pi_1,\ gl_0,\ gl_1,\ o_0,\ o_1)$
$+\Delta;[c]$	$CP(\Delta,\ \pi_0,\ 0\cdot\pi_1,\ gl_0,\ gl_1,\ o_0,\ o_1)$; $CI(gl_1,\ 1\cdot\pi_1,\ c,\ o_1)$
$\Delta_0;\Delta_1$	$CP(\Delta_0,\ 0\cdot\pi_0,\ 0\cdot\pi_1,\ gl_0,\ gl_1,\ o_0,\ o_1)$ $CP(\Delta_1,\ 1\cdot\pi_0,\ 1\cdot\pi_1,\ gl_0,\ gl_1,\ o_0,\ o_1)$
if $(b_0\rightarrow b_1)\ \Delta_0$ **else** Δ_1	**if** $(gl_0)\ \pi_0 = b_0$; **if** $(gl_1)\ \pi_1 = b_1$; $CP(\Delta_0,\ 1\cdot\pi_0,\ 1\cdot\pi_1,\ gl_0\wedge\pi_0,\ gl_1\wedge\pi_1,\ o_0,\ o_1)$; $CP(\Delta_1,\ 0\cdot\pi_0,\ 0\cdot\pi_1,\ gl_0\wedge\neg\pi_0,\ gl_1\wedge\neg\pi_1,\ o_0,\ o_1)$

while $(b_0\rightarrow b_1)\ \Delta$	**if** $(gl_0)\ \pi_0 = b_0$;
	if $(gl_1)\ \pi_1 = b_1$;
	while $((gl_0\wedge\pi_0)\vee(gl_1\wedge\pi_1))\ \{$
	  if $(gl_0\wedge\pi_0)\ 1\cdot\pi_0$ = **true**;
	  if $(gl_1\wedge\pi_1)\ 1\cdot\pi_1$ = **true**;
	  $CP(\Delta,\ 1\cdot1\cdot\pi_0,\ 1\cdot1\cdot\pi_1,\ gl_0\wedge\pi_0\wedge(1\cdot\pi_0),$
	  $gl_1\wedge\pi_1\wedge(1\cdot\pi_1)$, **Some** π_0, **Some** π_1);
	  if $(gl_0\wedge\pi_0)\ \pi_0 = b_0$;
	  if $(gl_1\wedge\pi_1)\ \pi_1 = b_1$;
	$\}$

$a_0\rightarrow a_1$	$CI(gl_0,\ \pi_0,\ a_0,\ o_0)$; $CI(gl_1,\ \pi_1,\ a_1,\ o_1)$
$-[$**if** $(b)\ c$ **else**$]\ \Delta$	**if** $(gl_0)\ \pi_0 = b$; $CI(gl_0\wedge\pi_0,\ 1\cdot\pi_0,\ c,\ o_0)$; $CP(\Delta,\ 0\cdot\pi_0,\ \pi_1,\ gl_0\wedge\neg\pi_0,\ gl_1,\ o_0,\ o_1)$
$+[$**if** $(b)\ c$ **else**$]\ \Delta$	**if** $(gl_1)\ \pi_1 = b$; $CI(gl_1\wedge\pi_1,\ 1\cdot\pi_1,\ c,\ o_1)$; $CP(\Delta,\ \pi_0,\ 0\cdot\pi_1,\ gl_0,\ gl_1\wedge\neg\pi_1,\ o_0,\ o_1)$
$-[$**if** $(b)]\ \Delta\ [$**else** $c]$	**if** $(gl_0)\ \pi_0 = b$; $CP(\Delta,\ 1\cdot\pi_0,\ \pi_1,\ gl_0\wedge\pi_0,\ gl_1,\ o_0,\ o_1)$; $CI(gl_0\wedge\neg\pi_0,\ 0\cdot\pi_0,\ c,\ o_0)$
$+[$**if** $(b)]\ \Delta\ [$**else** $c]$	**if** $(gl_1)\ \pi_1 = b$; $CP(\Delta,\ \pi_0,\ 1\cdot\pi_1,\ gl_0,\ gl_1\wedge\pi_1,\ o_0,\ o_1)$; $CI(gl_1\wedge\neg\pi_1,\ 0\cdot\pi_1,\ c,\ o_1)$

Fig. 3. Difference directed correlating program generation function CP.

Δ	$CP(\Delta,\ \pi_0,\ \pi_1,\ gl_0,\ gl_1,\ o_0,\ o_1)$
$-[\textsf{while}\ (b)]\ \Delta$	**if** $(gl_0)\ \pi_0 = b$**;** **if** $(gl_0 \wedge \pi_0)\ 1 \cdot \pi_0 =$ **true;** $CP(\Delta,\ 1 \cdot 1 \cdot \pi_0,\ \pi_1,\ gl_0 \wedge \pi_0 \wedge (1 \cdot \pi_0),\ gl_1,$ **Some** $\pi_0,\ o_1)$**;** **if** $(gl_0 \wedge \pi_0)\ \pi_0 = b$**;** **while** $(gl_0 \wedge \pi_0)$ **{** **if** $(gl_0 \wedge \pi_0)\ 1 \cdot \pi_0 =$ **true;** $CI(gl_0 \wedge \pi_0 \wedge (1 \cdot \pi_0),\ 1 \cdot 1 \cdot \pi_0,\ \Pi_0(\Delta),$ **Some** $\pi_0)$**;** **if** $(gl_0 \wedge \pi_0)\ \pi_0 = b$**; }**
$+[\textsf{while}\ (b)]\ \Delta$	**if** $(gl_1)\ \pi_1 = b$**;** **if** $(gl_1 \wedge \pi_1)\ 1 \cdot \pi_1 =$ **true;** $CP(\Delta,\ \pi_0,\ 1 \cdot 1 \cdot \pi_1,\ gl_0,\ gl_1 \wedge \pi_1 \wedge (1 \cdot \pi_1),\ o_0,$ **Some** $\pi_1)$**;** **if** $(gl_1 \wedge \pi_1)\ \pi_1 = b$**;** **while** $(gl_1 \wedge \pi_1)$ **{** **if** $(gl_1 \wedge \pi_1)\ 1 \cdot \pi_1 =$ **true;** $CI(gl_1 \wedge \pi_1 \wedge (1 \cdot \pi_1),\ 1 \cdot 1 \cdot \pi_1,\ \Pi_1(\Delta),$ **Some** $\pi_1)$**;** **if** $(gl_1 \wedge \pi_1)\ \pi_1 = b$**; }**

Fig. 4. Difference directed correlating program generation function CP (cont.).

Theorem 2 (`correlating_program` is sound). *Coq: correlating_program_sound*
*For all S, S_0, S_0', S_1, S_1', a diff Δ and a guard store G such that the judgments $S_0 \vdash \Pi_0(\Delta) \Downarrow_\square S_0'$ and $S_1 \vdash \Pi_1(\Delta) \Downarrow_\square S_1'$ hold ; S_0 and S_1 split S ; G_0 and G_1 split G, then there exists a store S' and a guard store G' such that $S, G \vdash CP(\Delta^{T,T'},\ 0,\ 1,$ **true**, **true**, **None**, **None**$) \Downarrow S', G'$ hold ; S_0' and S_1' split S ; and G_0 and G_1 split G'.*

Roughly speaking, this theorem states that the correlating program simulates the original and the modified programs correctly. As we are using big-step semantics, this theorem only characterizes terminating evaluations. In our opinion, a similar result can be proved in a small-step semantics to encompass non terminating evaluations. This is left as future work.

6 Implementation and Experiments

Implementation As said previously, we proved our algorithm within the Coq proof assistant. Our development is about 3,800 lines of Coq, 10% of which are definitions of the input and guarded languages, as well as the definition of what a correct correlating program is. The remaining lines are used for the algorithm and its soundness proof. The Coq development is available online at www.pps.univ-paris-diderot.fr/~thib/atva15/coq/.

This code is then extracted from Coq to OCaml. In addition to the extracted code, we wrote 1,000 lines of OCaml to parse the input language, to construct the syntactic difference and to print the correlating program in C syntax. One should notice that the generated C program semantics does not exactly match

the semantics of our formalized development. For example, our input language manipulates mathematical integers while the generated C program uses fixed-length machine integers. Albeit it was not done because outside the core of our work, we do not consider it would be conceptually difficult to integrate into our input language the semantics of the generated C code (e.g. 32 bits integers) because the generation algorithm manipulates the syntax of expressions abstractly. Moreover, the semantics of our language is compatible with the semantics of input C language expected by `dizy`. It should also be noted that the language presented in this paper has been slightly simplified for readability, while the actual tool and its formalization in Coq handle additional operators as well as a limited form of arrays.

To compute the structural syntactic differences, we aim at finding a minimal difference by an exploration of the space of mappings between abstract syntax trees. We start from the root nodes of the abstract syntax trees of the two programs. We then recursively descend along those trees, comparing at each level all possible differences and keeping the minimal one (using an heuristics that tries to minimize insertion or deletion of loops). We use some memoization to implement a weak form of dynamic programming. While this computation of the syntactic difference is not proven correct neither optimal in Coq, a mechanically verified checker dynamically ensures that the projections of the chosen structural difference are indeed the two input programs.

Regarding correlating points, which are an important aspect of Partush and Yavah's work and are essential for scalability, we have implemented a basic heuristics similar to theirs: we insert a correlating point in the generated program after two equivalent instructions of the input programs. However finding the best place to put them is orthogonal to the soundness of the correlating program generation itself which is our primary concern. Furthermore, we observed that in some instances, placing correlating points actually increases computation time due to increased complexity in the analyzer's sub-states. Therefore, we disabled them in our experiments.

Gap with real language like C Our language roughly corresponds to a small subset of the C language equipped with idealized integers and arrays. It would not be very difficult to handle C's "struct" and other kinds of type definition as our algorithm is only concerned about the control-flow of programs. "switch" and "for" constructs could also be easily integrated. Regarding pointers and "union" structure, we would have to ensure proper correspondence between pointed variables at abstract interpretation initialization as well as proper memory partitioning. `dizy` currently does not handle such issues. Regarding "goto" and "setjmp"/"longjmp", we are currently unable to handle them because that would require to encode any arbitrary displacement in the control-flow graph using guard variables. Even if such an encoding is possible in theory, there is little hope that an abstract interpretation could effectively infer interesting correlations out of the resulting correlating program.

Experiments To compare the *quality* of our correlating programs with the ones produced by `ccc`, a series of 23 examples (most between 10 and 20 lines long, with the exception of one around 140 lines long) were analyzed by `dizy`. While doing so, we found no instance where the correlating program produced by `ccc` enabled a more precise analysis than that permitted by the correlating program generated by `correlating_program`. On the contrary, we found several examples where `correlating_program` outperformed `ccc` (examples 6, 7 and 23). We also implemented a binary search algorithm in a sorted array, with the modified version introducing early loop exits. We can generate the correlating program and analyze it with `dizy` (by slightly modifying it to correctly handle read access to arrays). We also attempted to test more complex examples but were limited by `dizy`'s capabilities (e.g., no handling of C's bit-wise logical operations). All those tests are available online at www.pps.univ-paris-diderot.fr/∼thib/atva15/examples/. and can be reproduced. In practice, the computation of structural differences and the generation of the correlating program was almost instantaneous on our examples. Most of the computation time is spent in `dizy`.

7 Related Work

Formal treatment of equivalence between algorithms dates back to the 60s' (references [1, 2, 3] in [5]). However, the specific topic of semantic differences between program variants was only considered in 1990 by Susan Horwitz [6]. Like us, she focuses on intra-procedural differences and she compares a structured intermediate representation of the programs, a so-called Program Representation Graph, which is a graph mixing control and data flow information.

Strichman and Godlin pioneered the use of uninterpreted functions for doing inter-procedural analysis [15]. They use the CBMC bounded model checker to establish the equivalence between two variants of a C program, thus limiting analysis to bounded loops. In case the two programs semantically differs, a counter-example is proposed. The user needs to provide a list of program points when the two variants should be equivalent. Like us, they do not handle neither complex data structure nor pointers.

Lahiri et al. proposed the SymDiff tool to check equivalence between program and display semantic differences[8]. As they rely on the intermediate logical language Boogie, their approach can handle multiple imperative languages like C, C# or even x86 assembly through appropriate front-end. Their approach is inter-procedural, using uninterpreted functions or inlining to handle function calls. For C language, they handle pointers and arrays but assume there is no aliasing. The underlying technology is based on generation of verification conditions and use the Z3 SMT solver to solve them. In case differences are found, a counter-example is generated. The approach does not scale when the number of different intra-procedural paths is too important (beyond 1,000 paths).

Symbolic execution has also been used to characterize semantic differences. Person et al. [14] propose an inter-procedural analysis with two notions of equivalences: *functional equivalence* with same black-box behavior and *partition-effect*

equivalence where the program variants have the same sets of paths in their implementations. They are using uninterpreted functions to handle function calls. They need to store the analysis of the variants of analyzed programs. Yang et al. [16] also use symbolic execution but simultaneously analyze two variants of a program. Their approach considers unaffected program parts to increase the scalability and precision of the analysis. In both papers, the Java language is handled however proposed approaches are limited to bounded loops.

Gao et al. addressed the finding of semantic differences over binary files [4]. There are using a combination of Control Flow Graph, Symbolic Execution and Theorem proving to find similarities between basic blocks. However their approach is unsound due to approximations when they abstract x86 instructions.

An obvious reference for our work is the one of Partush and Yahav [12] as we built directly on top of it. They proposed an intra-procedural approach capable of handling unbounded loops over simple C programs without complex data structures or pointers. They propose both a way to interleave two variants of a program and an abstract interpretation technique focusing on establishing program equivalence or characterizing precise differences. In this paper, we show their approach is unsound and described a sound, mechanically checked, variant of their work. Partush and Yahav initial work has recently been improved with a technique to dynamically establish the best interleaving of the two programs during their analysis [13]. They use uninterpreted functions to analyze arrays or function calls. They are no longer using guards assertion to build interleaving programs, process that we have shown to be erroneous.

8 Conclusion

In this paper, we tackle the issue of characterizing the semantic differences between two versions of a program. We follow an approach similar to Partush and Yahav [12], building a correlating program representing the semantics of the two programs and then analyzing it using an Abstract Interpretation technique. This approach can handle unbounded loops. However, we show through counter-examples that the Partush and Yahav construction of correlating program is unsound for certain forms of loop and goto constructions.

We thus present an original and sound algorithm to build a correlating program from the structured syntactic difference of two programs. While we do not handle free form goto, we handle **break** and **continue** statements. We formalize and prove our algorithm within the Coq proof assistant, from which we extract our tool to ensure it is Correct by Construction. We compare our tool with the one of Partush and Yahav, observing that it is at least as precise as theirs.

This work only consider intra-procedural analysis of a simple language, without complex data structures, arrays or pointers. In the future, we would like to define intra-procedure semantic differences on other kinds of programs like functional, object-oriented or modular ones. We also want to have a wider and more structural view on a program, characterizing inter-procedural semantic differences through programmer level constructions like “adding a method to a class”

or "adding a parameter to a function". Our long-term objective is to propose a tool capable of describing semantic differences of a real-world development like the Linux kernel or the Firefox web browser.

References

1. Binkley, D.: Using semantic differencing to reduce the cost of regression testing. In: Software Maintenance, 1992. Proceerdings., Conference on. (Nov 1992) 41–50
2. Brumley, D., Poosankam, P., Song, D., Zheng, J.: Automatic patch-based exploit generation is possible: Techniques and implications. In: 2008 IEEE Symposium on Security and Privacy. SP '08, IEEE Computer Society (2008) 143–157
3. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM Press (1977) 238–252
4. Gao, D., Reiter, M.K., Song, D.: Binhunt: Automatically finding semantic differences in binary programs. In Chen, L., Ryan, M., Wang, G., eds.: Information and Communications Security. Volume 5308 of LNCS. Springer (2008) 238–255
5. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10) (October 1969) 576–580
6. Horwitz, S.: Identifying the semantic and textual differences between two versions of a program. SIGPLAN Not. **25**(6) (June 1990) 234–245
7. Jackson, D., Ladd, D.A.: Semantic diff: A tool for summarizing the effects of modifications. In: Proceedings of the International Conference on Software Maintenance. ICSM '94, Washington, DC, USA, IEEE Computer Society (1994) 243–252
8. Lahiri, S.K., Hawblitzel, C., Kawaguchi, M., Rebêlo, H.: SYMDIFF: A Language-agnostic Semantic Diff Tool for Imperative Programs. In: 24th International Conference on Computer Aided Verification. CAV'12, Springer (2012) 712–717
9. Lahiri, S.K., Vaswani, K., Hoare, C.A.R.: Differential static analysis: Opportunities, applications, and challenges. In: Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research. FoSER '10, New York, NY, USA, ACM (2010) 201–204
10. Leroy, X.: Formal verification of a realistic compiler. Communications of the ACM **52**(7) (2009) 107–115
11. The Coq development team: The Coq proof assistant reference manual. LogiCal Project. (2004) Version 8.0.
12. Partush, N., Yahav, E.: Abstract semantic differencing for numerical programs. In Logozzo, F., Fähndrich, M., eds.: Static Analysis Symposium. Volume 7935 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2013) 238–258
13. Partush, N., Yahav, E.: Abstract semantic differencing via speculative correlation. In: ACM International Conference on Object Oriented Programming Systems Languages & Applications. OOPSLA '14, New York, NY, USA, ACM (2014) 811–828
14. Person, S., Dwyer, M.B., Elbaum, S., Păsăreanu, C.S.: Differential symbolic execution. In: 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering. SIGSOFT '08/FSE-16, ACM (2008) 226–237
15. Strichman, O., Godlin, B.: Regression verification - a practical way to verify programs. In Meyer, B., Woodcock, J., eds.: Verified Software: Theories, Tools, Experiments. Volume 4171 of LNCS. Springer (2008) 496–501
16. Yang, G., Person, S., Rungta, N., Khurshid, S.: Directed incremental symbolic execution. ACM Trans. Softw. Eng. Methodol. **24**(1) (October 2014) 3:1–3:42