Difference Verification with Conditions

Dirk Beyer[1], Marie-Christine Jakobs[2,1], and Thomas Lemberger[1]

[1] LMU Munich, Munich, Germany
[2] TU Darmstadt, Department of Computer Science, Darmstadt, Germany

Abstract. Modern software-verification tools need to support development processes that involve frequent changes. Existing approaches for incremental verification hard-code specific verification techniques. Some of the approaches must be tightly intertwined with the development process. To solve this open problem, we present the concept of *difference verification with conditions*. Difference verification with conditions is independent from any specific verification technique and can be integrated in software projects at any time. It first applies a change analysis that detects which parts of a software were changed between revisions and encodes that information in a condition. Based on this condition, an off-the-shelf verifier is used to verify only those parts of the software that are influenced by the changes. As a proof of concept, we propose a simple, syntax-based change analysis and use difference verification with conditions with three off-the-shelf verifiers. An extensive evaluation shows the competitiveness of difference verification with conditions.

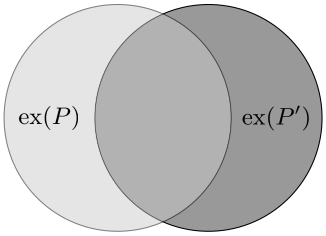
1 Introduction

Software changes frequently during its life-cycle: developers fix bugs, adapt existing features, or add new features. In agile development, software construction is an intrinsically incremental process. Every change to a working system holds a risk to introduce a new defect. Since software failures are often costly and may even endanger human lives, it is an integral part of software development to find potential failures and ensure their absence.

However, running a full verification after each change is inadequate: Changes rarely affect the complete program behavior. For example, consider program `absSum` (Fig. 1, middle). If the assignment of program variable `r` is changed in the else-branch at location 5 (`absSum`$^{\texttt{mod}}$, Fig. 1, right), only program executions that take that else-branch show different behavior. Program executions that take the if-branch (highlighted in gray) are not affected by the change. This is typical for program changes: A modified program P' exhibits some new or changed program executions compared to an original program P, but some executions also stay the same (Fig. 1, left). To ensure the safety of P', it is sufficient to inspect only the changed behavior $\text{ex}(P') \setminus \text{ex}(P)$.

Replication package available on Zenodo [12].

Funded in part by the Deutsche Forschungsgemeinschaft (DFG) – 418257054 (Coop) and 378803395 (GRK ConVeY).



```
0 r=0;
1 if(a<0)
2   while(a<0)
3     r=r-a;
4     a=a+1;
 else
5   r=a+a+1;
6   r=r/2;
```

```
0 r=0;
1 if(a<0)
2   while(a<0)
3     r=r-a;
4     a=a+1;
 else
5   r=a*(a+1);
6   r=r/2;
```

Fig. 1: Relation between program executions of original and modified program (left) and an example: Program `absSum` (middle) and its modified version `absSum`$^{\texttt{mod}}$ (right). The modification at location 5 is shown in blue. Program parts unaffected by the modification are highlighted in gray.

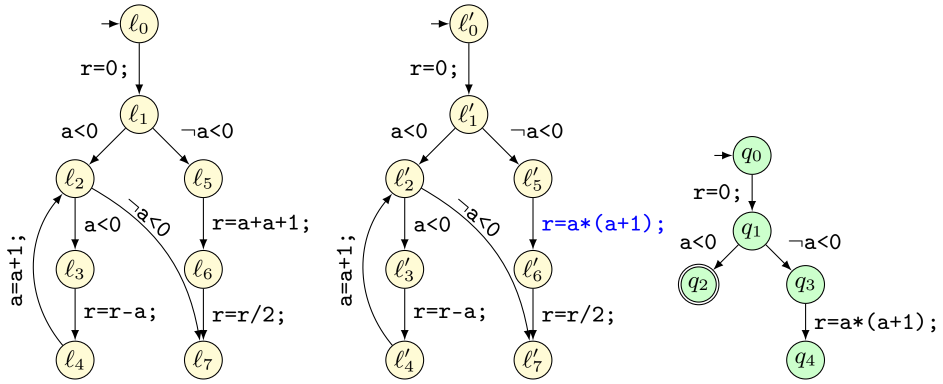
Many incremental verification approaches [39, 40] use this insight: Regression-test selection [62] tries to only execute those tests in a test suite that are relevant w.r.t. the change, and incremental formal verification techniques adapt existing proofs [33, 49, 53, 54], reuse intermediate results [16, 59], or skip the exploration of unchanged behavior [21, 47, 60, 61]. However, they (a) all focus on one fixed verification approach, (b) require a strong coupling between the original verification approach and the incremental technique, and (c) require an initial, full verification run. Often, this inflexibility makes an approach prohibitive.

As an alternative, we define the concept of *difference verification with conditions*: Given the original and the changed software, difference verification with conditions first identifies all executions that are affected by changes and encodes them in a condition, an exchange format already known from conditional model checking [10]—we call this first part diffCond. Then, a conditional verifier uses that condition to verify only the changed program behavior. For this step, any existing off-the-shelf verifier can be turned into a conditional verifier with the reducer-based approach [13].

Difference verification with conditions allows us to (a) use varying verification approaches for incremental verification, (b) automatically turn any existing verifier into an incremental verifier, and (c) skip an initial, costly verification run.

Contributions. We make the following contributions:

- We propose *difference verification with conditions*, which is an incremental verification approach that combines existing tools and approaches.
- We provide the algorithm diffCond, an integral part of difference verification with conditions, which outputs a description of the modified execution paths in an exchangeable condition format. We also prove its correctness.
- We implemented diffCond in the verification framework CPAchecker and combined it with existing verifiers to construct difference verifiers.
- To study the effectiveness and efficiency of difference verification with conditions, we performed an extensive evaluation on more than 10 000 C programs.
- diffCond and all our data are available for replication and to construct further difference verifiers (see Sect. 7).



(a) Original program (b) Modified program (c) Condition

Fig. 2: CFA of `absSum` (Fig. 1), CFA of `absSum`$^{\texttt{mod}}$, and a condition that describes the common executions of both programs, as created by our approach

2 Background

Programs. For ease of presentation, we consider imperative programs with deterministic control-flow, which execute statements from a set *Ops*. Our implementation supports C programs. Following literature [8, 9, 30], we model programs as control-flow automata.

Definition 1. *A* control-flow automaton *(CFA) $P = (L, \ell_0, G)$ consists of*
- *a set L of program locations with initial location $\ell_0 \in L$, and*
- *a set $G \subseteq L \times Ops \times L$ of control-flow edges.*

CFA P is deterministic *if $(\ell, op, \ell'), (\ell, op, \ell'') \in G \Rightarrow \ell' = \ell''$.*

Figure 2 shows the CFA of the example program `absSum` from Fig. 1. A sequence $\ell_0 \overset{op_1}{\to} \ell_1 \cdots \overset{op_n}{\to} \ell_n$ is a *syntactical path* through CFA $P = (L, \ell_0, G)$, if $\forall i \in [1, n] : (\ell_{i-1}, op_i, \ell_i) \in G$. We rely on standard operational semantics and model a program state by a pair of (1) the program counter, whose value refers to a program location in the CFA, and (2) a concrete data state c, whose shape we do not further specify [8]. We denote the set of all concrete data states as C. The function $sp_{op} : C \to 2^C$ describes the possible effects of operation $op \in Ops$ on concrete data state $c \in C$. Based on this, a sequence $(\ell_0, c_0) \overset{op_1}{\to} (\ell_1, c_1) \cdots \overset{op_n}{\to} (\ell_n, c_n)$ is a *program path* through CFA $P = (L, \ell_0, G)$, if $\ell_0 \overset{op_1}{\to} \ell_1 \cdots \overset{op_n}{\to} \ell_n$ is a syntactical path through P and $\forall i \in [1, n] : c_i \in sp_{op_i}(c_{i-1})$. We denote the set of all program paths by $\text{paths}(P)$. *Program executions* are derived from program paths. If $p = (\ell_0, c_0) \overset{op_1}{\to} (\ell_1, c_1) \cdots \overset{op_n}{\to} (\ell_n, c_n)$ is a program path, then $\text{ex}(p) = c_0 \overset{op_1}{\to} c_1 \cdots \overset{op_n}{\to} c_n$ is a program execution. The executions of a program P are defined as $\text{ex}(P) := \{ex(p) \mid p \in \text{paths}(P)\}$.

Conditions. A condition describes which program executions were already verified, e.g., in a previous verification run. We use automata to represent conditions and use accepting states to identify already verified executions [13].

Definition 2. *A* condition $A = (Q, \delta, q_0, F)$ *consists of:*

- *a finite set Q of states,*
- *a transition relation $\delta \subseteq Q \times Ops \times Q$ ensuring $\forall (q, op, q') \in \delta : q \in F \Rightarrow q' \in F$,*
- *the initial state $q_0 \in Q$, and a set $F \subseteq Q$ of accepting states.*[3]

The goal of `absSum` (left program in Fig. 2) is to compute $r = \sum_{i=0}^{|a|}$. However, the original program is buggy: In location ℓ_5, it must compute the product of a and $a+1$, not the sum. The fixed program is shown in the middle of Fig. 2—the fix is highlighted in blue. The original and modified version of the program only differ in the else-branch. If we assume that the original program was already verified, we know that program executions passing through the if-branch have already been verified and do not need to be considered during a reverification. In contrast, executions that pass through the else-branch and reach the modified statement must be verified. The condition shown on the right of Fig. 2 encodes this insight. Program executions that pass through the if-branch ($a < 0$) lead to the accepting state q_2—we say they are covered by the condition. In contrast, program executions that pass through the else-branch ($\neg a < 0$) never reach q_2 —they are not covered by the condition, and must be analyzed.

Definition 3. *A condition* $A = (Q, \delta, q_0, F)$ covers an execution $\pi = c_0 \overset{op_1}{\to} c_1 \cdots \overset{op_n}{\to} c_n$ *if there exists an index* $k \in [0, n]$ *and a run* $\rho = q_0 \overset{op_1}{\to} q_2 \cdots \overset{op_k}{\to} q_k$, *s.t.* $q_k \in F$ *and* $\forall i \in [1, k] : (q_{i-1}, op_i, q_i) \in \delta$.

Next, we introduce a simple and efficient way to systematically compute a condition that covers the common executions of an original and a modified program.

3 Component diffCond for Modular Construction

The ultimate goal of difference verification with conditions is to speed up reverification of modified programs. To achieve this goal, we aim at ignoring unmodified program behavior during verification. Conditions are a well-fitting format to describe the unmodified program behavior. However, to benefit from difference verification with conditions, the construction of such conditions must be efficient, i.e., consume only a small portion of the overall execution time of the verification. Therefore, we use a syntactic approach to compute the condition, diffCond (Alg. 1), which is linear in time regarding the size of the modified program.

diffCond gets as input the original program P and the modified program P'. In lines 1 to 11, diffCond traverses the modified and the original program in parallel, stops traversal if the original and the modified program differ, and remembers the edge that differs in the modified program.

It uses three data structures: Set $E \subseteq L \times L' \times Ops \times L \times L'$ stores all compared edges (ℓ_1, op, ℓ_2) and (ℓ'_1, op, ℓ'_2) that are equal in both programs. These edges are

[3] In general [10, 13] the transition relation of a condition also specifies assumptions on the program states. Since difference verification with conditions requires no assumptions on the program states, we omit this additional characteristic.

Algorithm 1 DIFFCOND(P, P')

```
Input: CFA $P = (L, \ell_0, G)$    // original program
Input: CFA $P' = (L', \ell'_0, G')$    // modified program
Output: $A = (Q, \delta, q_0, F)$    // difference condition
Variables: Set $E \subseteq L \times L' \times Ops \times L \times L'$ of composite CFA edges equal in the original
    and the modified program, set $D \subseteq L \times L' \times Ops \times L'$ of CFA edges that differ in
    the modified program, set $waitlist \subseteq L \times L'$ of program locations in original and
    modified program for which to compare outgoing edges.

▷ Change detection
$E := \emptyset$; $D := \emptyset$
$waitlist := \{(\ell_0, \ell'_0)\}$
while $waitlist \neq \emptyset$ do
    pop $(\ell_1, \ell'_1)$ from $waitlist$
    for each $(\ell'_1, op, \ell'_2) \in G'$ do
        if $\neg\exists \ell_2 \in L : (\ell_1, op, \ell_2) \in G$ then
            $D := D \cup \{((\ell_1, \ell'_1), op, \ell'_2)\}$
        else
            $E := E \cup \{((\ell_1, \ell'_1), op, (\ell_2, \ell'_2))\}$
            if $(\cdot, \cdot, (\ell_2, \ell'_2)) \notin E$ then
                $waitlist := waitlist \cup \{(\ell_2, \ell'_2)\}$

▷ Condition Generation
$Q := \{q \mid \exists(\cdot, \cdot, q) \in D\}$
$waitlist := Q$
while $waitlist \neq \emptyset$ do
    pop $q'$ from $waitlist$
    for each $(q, op, q') \in E \cup D$ with $q \notin Q$ do
        $Q := Q \cup \{q\}$
        $waitlist := waitlist \cup \{q\}$
if $Q = \emptyset$ then
    ▷ No difference edges, automaton always accepts
    return $(\{(\ell_0, \ell'_0)\}, \emptyset, (\ell_0, \ell'_0), \{(\ell_0, \ell'_0)\})$
else
    $F := \{q' \mid \exists(q, op, q') \in E \wedge q \in Q \wedge q' \notin Q\}$
    $Q := Q \cup F$
    $\delta := \{(q, op, q') \in E \cup D \mid q, q' \in Q \wedge q \notin F\}$

    return $(Q, \delta, (\ell_0, \ell'_0), F)$
```

called *standard edges*. They are stored in the composite form $((\ell_1, \ell'_1), op, (\ell_2, \ell'_2))$. Set $D \subseteq L \times L' \times Ops \times L'$ stores all edges (ℓ'_1, op, ℓ'_2) of the modified program P' that represent a change from the original program P at ℓ_1, called *difference edges*. They are stored in the form $((\ell_1, \ell'_1), op, \ell'_2)$. Set $waitlist \subseteq L \times L'$ stores all pairs of program locations (ℓ_1, ℓ'_1) for which a program path with the same syntactic structure exist in P and P', and for which no outgoing edges have been considered yet. Initially, E and D are empty—no edges were checked so far, and the algorithm

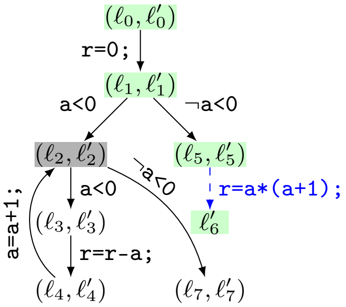


Fig. 3: Parallel composition of `absSum` and `absSum`$^{\texttt{mod}}$ as computed by diffCond

starts at the two initial program locations, i.e., *waitlist* $= \{(\ell_0, \ell'_0)\}$ (lines 1 and 2). As long as *waitlist* contains program locations, the algorithm picks one of them, here depicted as (ℓ_1, ℓ'_1) (line 4). It considers all outgoing edges (ℓ'_1, op, ℓ'_2) of ℓ'_1 in the modified program. If the same operation *op* does not exist at any outgoing edge of ℓ_1, it is considered to be changed and the difference edge $((\ell_1, \ell'_1), op, \ell'_2)$ is stored in D before continuing with the next state in *waitlist*. However, if the same operation *op* exists at an outgoing edge (ℓ_1, op, ℓ_2), it is considered to be equal and the standard edge $((\ell_1, \ell'_1), op, (\ell_2, \ell'_2))$ is stored in E before continuing with the next state in *waitlist*. To this end, diffCond explores the syntactical composition of the original and modified program. In addition, if the tuple (ℓ_2, ℓ'_2) of locations has not been detected before (line 10), it is added to the waitlist for further exploration. Figure 3 shows the graph built from edges E (black) and D (blue and dashed) when executing diffCond on `absSum` and `absSum`$^{\texttt{mod}}$.

To compute the condition, we first determine the condition's states. Lines 12 to 18 compute all nodes that can reach a successor of a difference edge. Figure 3 highlights these nodes in green. Nodes that are not discovered in lines 12–18 cannot lead to a difference edge and, thus, not to different program behavior. Consequently, undiscovered nodes that are successors of nodes discovered in lines 12–18 become final states (line 23). Figure 3 highlights these nodes in gray (only node (ℓ_2, ℓ'_2)). The union of discovered and final states become our condition states. To complete the construction, we use the pair of initial program locations as the initial state (ℓ_0, ℓ'_0) and add to the transition relation all transitions from E and D that connect condition states. Figure 2c shows the condition created from Fig. 3.

Finally, note that lines 19–21 handle the special case that the set D of difference edges is empty, thus resulting in $Q = \emptyset$ in line 19. The set D is empty if the original and the modified program only differ in the names of their program locations [4] or if the modified program is empty $((\ell'_0, \cdot, \cdot) \notin G')$. In both cases, all executions of the modified program are covered by the executions of the original program. As a result, the condition covers all executions: its only state is both initial and accepting state, and the condition has no transitions.

The purpose of algorithm diffCond is to compute a condition that supports skipping unchanged behavior during reverification of a modified program.

[4] In practice, this can happen if empty lines are added or removed from the program.

To still have a sound reverification, the produced condition must not cover executions that do not occur in the original program. The following theorem states this property of algorithm diffCond.

Theorem 1. *Let $P = (L, \ell_0, G)$ and $P' = (L', \ell'_0, G')$ be two CFAs.* diffCond*(P,P') does not cover any execution from* $\mathrm{ex}(P') \setminus \mathrm{ex}(P)$.

Proof. Assume $\mathrm{ex}(P') \setminus \mathrm{ex}(P) \neq \emptyset$. Hence, $\text{diffCond}(P, P') = (Q, \delta, q_0, F)$ is returned in line 27. Let $(Q, \delta, q_0, F) = A$, let $\pi = c_0 \overset{op_1}{\rightarrow} c_1 \cdots \overset{op_n}{\rightarrow} c_n \in \mathrm{ex}(P') \setminus \mathrm{ex}(P)$, and let $\rho = q_0 \overset{op_1}{\rightarrow} q_1 \cdots \overset{op_k}{\rightarrow} q_k$ be a run through A, s.t. $0 \leq k \leq n$ and $\forall 1 \leq i \leq k : (q_{i-1}, op_i, q_i) \in \delta$. By construction, (1) $q_0 \notin F$, (2) $\forall 1 \leq i < k : (q_{i-1}, op_i, q_i) \in E \wedge q_i \notin F$, and (3) $(q_{k-1}, op_k, q_k) \in E \cup D$. We need to show that $q_k \notin F$. Case $k = 0$ follows from (1).

Next, consider the case $k = n$. If $(q_{k-1}, op_k, q_k) \in E$, by construction there exists syntactical path $sp = \ell_0 \overset{op_1}{\rightarrow} \ell_2 \cdots \overset{op_n}{\rightarrow} \ell_n$ in P and due to program semantics, $\pi \in \mathrm{ex}(P)$. Since $\pi \in \mathrm{ex}(P') \setminus \mathrm{ex}(P)$, we infer $(q_{k-1}, op_k, q_k) \in D$ and thus $q_k \notin F$.

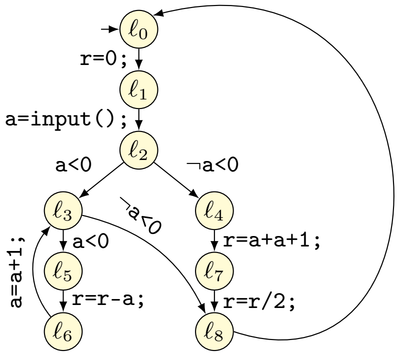
Finally, consider the case $k < n$. If $(q_{k-1}, op_k, q_k) \in D$, we infer $q_k \notin F$. Assume $(q_{k-1}, op_k, q_k) \in E$. By construction, there exists a syntactical path $sp = \ell_0 \overset{op_1}{\rightarrow} \ell_2 \cdots \overset{op_k}{\rightarrow} \ell_k$ in program P and a syntactical path $sp' = \ell'_0 \overset{op_1}{\rightarrow} \ell'_2 \cdots \overset{op_k}{\rightarrow} \ell'_k$ in program P', s.t. $\forall 0 \leq i \leq k : q_i = (\ell_i, \ell'_i)$. Let $\ell_0 \overset{op_1}{\rightarrow} \ell_2 \cdots \overset{op_k}{\rightarrow} \ell_k \overset{op_{k+1}}{\rightarrow} \ell_{k+1} \cdots \overset{op_m}{\rightarrow} \ell_m$ be an extension of the syntactical path sp s.t. $m = n$ or $(\ell_m, op_{m+1}, \cdot) \notin G$. Due to program semantics and $\pi \in \mathrm{ex}(P') \setminus \mathrm{ex}(P)$, we conclude $k \leq m < n$. Due to program semantics, P' being deterministic, and $\pi \in \mathrm{ex}(P')$, there exists an extension $\ell'_0 \overset{op_1}{\rightarrow} \ell'_2 \cdots \overset{op_k}{\rightarrow} \ell'_k \overset{op_{k+1}}{\rightarrow} \ell'_{k+1} \cdots \overset{op_m}{\rightarrow} \ell'_m$ of the syntactical path sp'. By construction, $\forall 1 \leq i \leq m : ((\ell_{i-1}, \ell'_{i-1}), op_i, (\ell_i, \ell'_i)) \in E$ and there exists $((\ell_m, \ell'_m), op_{m+1}, \cdot) \in D$. Hence, $\forall 0 \leq i \leq m : (\ell_{i-1}, \ell'_{i-1}) \in Q \setminus F$. Since $q_k = (\ell_k, \ell'_k)$ and $k \leq m$, $q_k \notin F$.

Theoretical Limitations. The effectiveness of difference verification with conditions depends on the amount of program code potentially affected by a change, which is determined by the diffCond component. diffCond only excludes program parts that cannot be syntactically reached from a program change. Therefore, difference verification is ineffective if some initial variable assignments at the very beginning of the program or some global declarations change. Moreover, the structure of a program strongly influences the effectiveness of difference verification. For example, programs like `absSum`$^\infty$ (Fig. 4) that mainly consist of a loop are problematic. Program `absSum`$^\infty$ (Fig. 4) is similar to `absSum`, but has an additional, outer loop that dominates the program. So when location ℓ_7 is changed in `absSum`$^\infty$, difference verification with conditions can only exclude the if-branch for the very first iteration of the outer loop. Thereafter, the change in location ℓ_7 may propagate into the if-branch.

In contrast, difference verification with conditions can be effective on programs that allow the exclusion of program parts, e.g., if the program is modular and, thus, consists of multiple, loosely coupled parts. Examples for modularity are the *strategy* design pattern, object-oriented software, or software applications with multiple program features.

```
0 while (1)
1   r=0;
2   a=input();
3   if(a<0)
4     while(a<0)
5       r=r-a;
6       a=a+1;
    else
7     r=a+a+1;
8     r=r/2;
```

(a) Program code



(b) CFA of program

Fig. 4: Example program absSum$^\infty$ with loop dominating the whole program

When designing our experiments, we will consider these limitations of difference verification with conditions. Before we get to our experiments, we must describe the modular composition of the diffCond component with a verifier, which specifies the difference verifier.

4 Modular Combinations with Existing Verifiers

The diffCond algorithm can be combined with any off-the-shelf conditional verifier [10] to produce a difference verifier in a modular way. The goal of a difference verifier is to verify only modified program paths. To this end, it first uses diffCond to discover potentially modified program paths and then runs a conditional verifier to explore only those paths identified by diffCond. Figure 5 shows the construction template for difference verification with conditions. diffCond gets the original and modified program as input and encodes the modified paths in a condition. The constructed condition is forwarded to a conditional verifier, which uses the condition to restrict its analysis of the modified program to those paths that are not covered by the condition (i.e., the modified paths). Based on this template, we can construct difference verifiers from arbitrary conditional verifiers. Moreover, we can construct difference verifiers from non-conditional verifiers by using the concept of reducer-based conditional verifiers [13]. The idea of a reducer-based conditional verifier is shown on the right of Fig. 5. To turn an arbitrary verifier into a conditional one, a reducer-based conditional verifier puts a preprocessor (called reducer) in front of the verifier. The reducer gets a program and a condition and outputs a new, residual program that represents the program paths not covered by the condition. A full verification

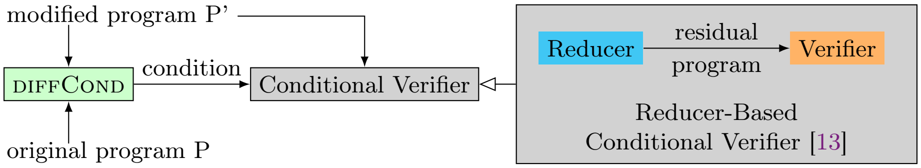


Fig. 5: diffCond + conditional verifier = difference verifier

of this residual program is then equivalent to a conditional verification of the original program with the produced condition. However, note that the existing reducers are designed for model checkers and do not necessarily work with other verification technologies like deductive verifiers.

In this paper, we transform three verifiers into difference verifiers: CPA-Seq, UAutomizer, and Predicate. The first two are the best verifiers from SV-COMP 2020 [5], and the third is a predicate-abstraction approach. We use the off-the-shelf verifiers CPA-Seq and UAutomizer as non-conditional verifiers and thus add a reducer, while we use Predicate as conditional verifier. Since a difference verifier can now be built from any off-the-shelf verifier, we can also combine difference verification with other incremental verification techniques. As an example, we can use precision reuse [16]. This technique is implemented in CPAchecker [16] and UAutomizer [49] and can be used with the previously mentioned approaches. Next we explain the technologies of the selected verifiers.

CPA-Seq uses several different strategies from the CPAchecker verification framework [6, 11, 14]. CPA-Seq first analyzes different features of the program under verification. The program features considered are: recursion, concurrency, occurrence of loops, and occurrence of complex data types like pointers and structs. Based on these features, CPA-Seq uses one of five different verification techniques (cf. [6]). For non-recursive, non-concurrent programs with a non-trivial control flow, CPA-Seq uses a sequential combination of four different analyses: It uses value analysis with and without Counterexample-guided Abstraction Refinement (CEGAR) [24], a predicate analysis similar to Predicate, and k-induction with invariant generation [7]. Invariants are generated by numerical and predicate analyses and are forwarded to the k-induction analysis.

UAutomizer is the automata-based approach from the Ultimate verification framework [29, 31]. It uses a CEGAR approach to successively refine an over-approximation of the error paths, which is given in form of automata. In each refinement step, a generalization of an infeasible error path is excluded from the over-approximation. The generalization of the error path is described by a Floyd-Hoare automaton [31], which assigns Boolean formulas over predicates to its states. The predicates are obtained via interpolation along the infeasible error path [43].

Predicate is the predicate-abstraction approach from the CPAchecker framework [14] with adjustable-block encoding (ABE) [15]. ABE is instructed to abstract at loop heads only. CEGAR together with lazy refinement [34] and interpolation [32] determines the necessary set of predicates.

PrecisionReuse is a competitive incremental approach that avoids recomputing the required abstraction level [16]. The idea is to start with the abstraction level determined in a previous verification run. To this end, it stores and reuses the precision, which describes the abstraction level, e.g., the set of predicates to be tracked. We use the version as implemented in CPAchecker.

5 Evaluation

We systematically evaluate our proposed approach along the following claims:

Claim 1. Difference verification with conditions can be more effective than a full verification. *Evaluation Plan:* For all verifiers, we compare the number of tasks solved by difference verification with conditions and by the pure verifier.

Claim 2. Difference verification with conditions is more effective when using multiple verifiers. *Evaluation Plan:* We compare the number of tasks solved by each difference verifier with the union of tasks solved by all difference verifiers.

Claim 3. Difference verification with conditions can be more efficient than a full verification. *Evaluation Plan:* For all verifiers, we compare the run time of difference verification with conditions and of the pure verifier.

Claim 4. The run time of difference verification with conditions is dominated by the run time of the verifier. *Evaluation Plan:* We relate the time for verification to the time required by the diffCond algorithm and the reducer.

Claim 5. Difference verification with conditions can complement existing incremental verification approaches. *Evaluation Plan:* We compare the results of difference verification with conditions with the results of precision reuse [16], a competitive incremental verification approach.

Claim 6. Combining difference verification with conditions with existing incremental verification approaches can be beneficial. *Evaluation Plan:* We compare the results of difference verification with the results of a combination of difference verification with conditions and precision reuse.

5.1 Experiment Setup

Computing Environment. We performed all experiments on machines with an Intel Xeon E3-1230 v5 CPU, 3.4 GHz, with 8 cores each, and 33 GB of memory, running Ubuntu 18.04 with Linux kernel 4.15. We limited each analysis run to 15 GB of memory, a time limit of 900 s, and 4 CPU cores. To enforce these limits, we ran our experiments with BenchExec [17], version 2.3.

Verifiers. For our experiments, we use the software verifiers CPA-Seq[5] [6, 14] and UAutomizer[6] [29, 31] as submitted for SV-COMP 2020, and CPAchecker [14, 15]

[5] https://gitlab.com/sosy-lab/sv-comp/archives-2020/-/raw/master/2020/cpa-seq.zip

[6] https://gitlab.com/sosy-lab/sv-comp/archives-2020/-/raw/master/2020/uautomizer.zip

in revision 32864[7]. CPA-Seq and UAutomizer are used as verifiers. CPAchecker provides the verifier Predicate, but also the new diffCond component and the Reducer component for reducer-based conditional verification. The difference verifier based on Predicate is realized as a single run. In contrast, the difference verifiers based on CPA-Seq and UAutomizer are realized as composition of two separate runs. The first run executes the diffCond algorithm followed by the reducer to generate the residual program. It is only executed once per task, i.e., the same residual programs are given to CPA-Seq and UAutomizer. In a second run, CPA-Seq and UAutomizer, respectively, verify the residual program. To deal with residual programs, we increased the Java stack size for CPA-Seq and UAutomizer.

Existing Incremental Verifier. We use Predicate with precision reuse [16].

Verification Tasks. We use verification tasks from the public repository `sv-benchmarks` (tag `svcomp20`)[8], which is the most diverse, largest, and well-established collection of verification tasks. Since difference verification with conditions is an incremental verification approach, we require different program versions. We searched the benchmark repository for programs that come with multiple versions and for which at least one version is hard to solve, i.e., at least one of the three considered verifiers takes more than 100 s for verification of that version, but is successful. From these programs, we arbitrarily picked the following: `eca05` and `eca12` (event-condition-action systems, both have 10 versions each), `gcd` (greatest common divisor computation, has 4 versions), `newton` (approximation of sine, has 24 versions), `pals` (leader election, has 26 versions), `sfifo` (second-chance FIFO replacement, has 5 versions), `softflt` (a software implementation of floats, has 5 versions), `square` (square-root computation, has 8 versions), and `token` (a communication protocol, has 28 versions). Unfortunately, all of these programs are specialized implementations with a single purpose. Thus, their implementation is strongly coupled and any reasonable program change affects the complete program. As explained before, this prohibits effective difference verification with conditions.

To get benchmark tasks that instead contain independent program parts, we create new combinations from the selected programs. We choose two programs, e.g., `eca05` and `token`. We then combine these two programs according to the following scheme: We create a new program with all declarations and definitions of both original programs, but a new main function. This new main function randomly calls the main function of one of the two original programs. Name clashes are resolved via renaming. Figure 6 shows the conceptual structure of each program created through this combination. For our experiments, we consider the following combinations of programs: (1) eca05+token, (2) gcd+newton, (3) pals+eca12, (4) sfifo+token, (5) square+softflt. To create different versions of our combinations, we replace one of the two program parts with a different version of that part. For example, to get a different

[7] https://gitlab.com/sosy-lab/software/cpachecker/-/tree/230d2ca5
[8] https://github.com/sosy-lab/sv-benchmarks/tree/svcomp20

```
extern int __VERIFIER_nondet_int();
int main1() { /* main method of task 1 ... */ }
/* other definitions of task 1 ... */
int main2() { /* main method of task 2 ... */ }
/* other definitions of task 2 ... */
int main() {
  if (__VERIFIER_nondet_int())
    main1();
  else
    main2();
}
```

Fig. 6: Conceptual example of combination of verification tasks

version of the original program `eca05+token`, we change the version of the `eca05` part or the `token` part, but never both.

With this procedure, we get a large amount of different versions of our program combinations. For our evaluation, we consider each pair (O, N) of versions O and N of program combinations that fulfills the following two conditions: (1) N reflects a change, i.e., the two programs are different. (2) Version O, version N, or both versions are bug-free. This ensures that verification and difference verification can only find the same bugs. With this construction of benchmark tasks for incremental verification we get a total of 10 426 tasks that we use in our experiments.

5.2 Experimental Results

Claim 1 (Difference verification with conditions more effective). Table 1 gives an overview of our experimental results. Each column represents one task set. The rows refer to verifiers, i.e., pure verifiers (X) and difference verifiers (X^{Δ}). The last two rows are the union of the results of all three verifiers. For each task set and verifier, the table provides the number of tasks for which the verifier finds a proof (✓), finds a bug (!), and only the difference verifier gives a conclusive answer (★). It also shows the number of tasks (◆) that cannot be solved. Neither the pure nor the difference verifiers reported incorrect results.

The table shows that for each verifier there exist task sets on which the number of solved correct tasks (✓) is higher for the difference verifier. Looking at columns ★, we observe that typically there exist tasks that only the difference verifier can solve. Thus, this shows that our new difference verification with conditions can be more effective.

Difference verification with conditions is not always more effective. Especially, CPA-Seq$^{\Delta}$ and UAutomizer$^{\Delta}$ sometimes perform worse. For example, CPA-Seq$^{\Delta}$ finds significantly less bugs than CPA-Seq for `eca05+token`. The reason for this is the residual program constructed by the reducer, which is necessary to turn

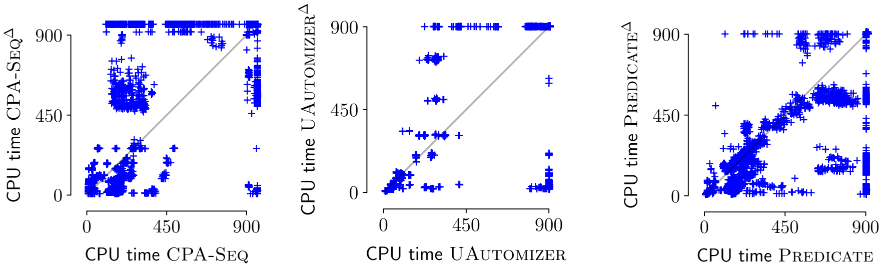
Table 1: Experimental results for Predicate, CPA-Seq and UAutomizer, as pure verifiers (X) and difference verifiers (X^{Δ}) showing how many correct tasks (✓) and tasks with a bug (!) are solved, how many tasks are only solved by the difference verifier (★) and which are too hard to solve (◆)

	eca05+token (3 640)				gcd+newton (1 924)				pals+eca12 (2 750)				sfifo+token (1 872)				square+softflt (240)			
	✓	!	★	◆	✓	!	★	◆	✓	!	★	◆	✓	!	★	◆	✓	!	★	◆
Predicate$^{\Delta}$	1 447	999	451	1 194	48	572	48	1 304	15	55	20	2 680	655	494	98	723	81	75	70	84
Predicate	1 080	944		1 616	0	572		1 352	0	50		2 700	558	507		807	33	53		154
CPA-Seq$^{\Delta}$	966	671	350	2 003	48	572	48	1 304	183	50	233	2 517	480	390	108	1 002	61	69	61	110
CPA-Seq	755	1 268		1 617	0	572		1 352	0	0		2 750	372	619		881	0	75		165
UAutomizer$^{\Delta}$	270	260	270	3 110	16	0	16	1 908	0	0	0	2 750	349	234	112	1 289	61	45	49	134
UAutomizer	0	325		3 315	0	520		1 404	0	48		2 702	341	258		1 273	44	57		139
All$^{\Delta}$	1 527	999	448	1 114	48	572	48	1 304	183	95	228	2 472	655	494	98	723	81	75	40	84
All	1 080	1 295		1 265	0	572		1 352	0	50		2 700	558	626		688	55	75		110

CPA-Seq into the required conditional verifier. The created residual programs, on which the off-the-shelf verifiers run, have a different structure than the original program. They make heavy use of goto statements and deeply nested branching structures. While semantically equivalent, this can have unexpected effects on analyses: In the case of the tasks in `eca05+token`, CPA-Seq was not able to detect required information about loops and thus aborts its verification. Note that this is not a direct issue of difference verification with conditions, but an orthogonal issue. To fix the problem, verification tools must be improved to better deal with the generated residual programs or the structure of the residual program must be improved. Despite of the problem with residual programs, difference verification can solve many tasks that a full verification run cannot solve.

Since Predicate is already a conditional model checker, Predicate$^{\Delta}$ does not suffer from the residual program problem. Thus, the effectiveness of difference verification with conditions becomes even more obvious when comparing Predicate with Predicate$^{\Delta}$. For the first three task sets, Predicate$^{\Delta}$ solves all tasks that Predicate solves plus a significant amount of additional tasks that Predicate cannot solve. For the last two task sets Predicate$^{\Delta}$ fails to solve a few tasks that Predicate can solve. However, Predicate$^{\Delta}$ still solves more tasks in total. One reason for this is that the predicate abstraction used by Predicate may compute different predicates (due to a slightly different exploration of the state space), which may result in a more expensive abstraction, if the explored state-space looks different. For some tasks, these different predicates may be less suited to solve the task and thus require more time, which results in the analysis hitting the time limit. Typically, we observe this phenomenon when Predicate is expensive already (in our experiments, when it takes at least 700 s). While for complicated tasks with large changes, difference verification may produce worse results, Predicate$^{\Delta}$ is still more effective than Predicate in all categories.

Claim 2 (Better with several verifiers). To study the usefulness of using several verifiers in difference verification, we look at the tasks solved by the three difference verifiers together. We observe that Predicate$^{\Delta}$ solves the most tasks in all task sets except for `pals+eca12`, in which CPA-Seq$^{\Delta}$ is better. Moreover,



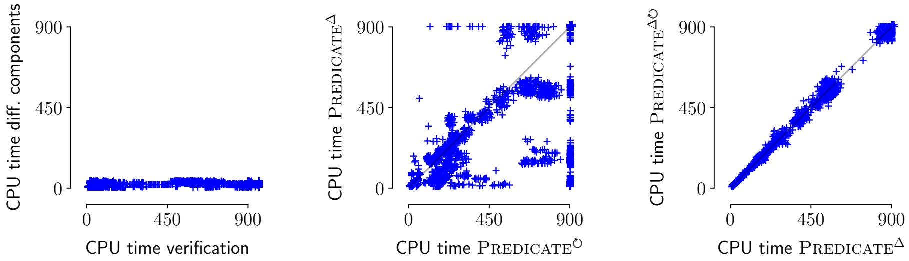
(a) CPA-Seq, CPA-Seq$^{\Delta}$ (b) UAutomizer, UAutomizer$^{\Delta}$ (c) Predicate, Predicate$^{\Delta}$

Fig. 7: CPU time (in s) of full verification vs. difference verification, per task

when looking at All$^{\Delta}$, which takes the union of all results, we observe that for `eca05+token` multiple tasks without a property violation exist that cannot be solved by the best difference verifier of this task set (Predicate$^{\Delta}$). Thus, the difference verification is more effective when using several verifiers.

Claim 3 (Difference verification with conditions more efficient). We compare the run times of the verifiers with the run times of the difference verifiers. For all three verifiers, the scatter plots in Fig. 7 show the CPU time required to check a task without (x-axis) and with difference verification (y-axis). If a task was not solved, because the verifier either runs out of resources or encountered an error, we assume the maximum CPU time of 900 s. Figures 7a and 7b compare the two non-conditional verifiers CPA-Seq and UAutomizer, for which we use the reducer-based conditional verifier approach. For a significant number of tasks (below diagonal), the difference verifier is faster than the respective verifier CPA-Seq and UAutomizer, and the tasks on the right edge can only be solved by the difference verifier. There are tasks for which difference verification is slower (above diagonal). Note that the problem is the residual program, not our approach. For example, many tasks located at the upper edge do not represent timeouts of the difference verification, but failures of the verifier caused by the structure of the residual program. Figure 7c compares the conditional verifier Predicate. For the majority of tasks, the CPU time required by Predicate$^{\Delta}$ is equal to or less than the time required by Predicate (tasks below the line). Moreover, there are only few tasks for which Predicate$^{\Delta}$ is slower than Predicate (tasks above the line). The reason for this slow-down is most likely the computation of worse predicates (see Claim 1). To sum up, difference verification with conditions can successfully increase efficiency.

Claim 4 (Verifier dominates run time). We aim to show that the diffCond component and the residual program construction (in the reducer-based approach to construct conditional verifiers) require a negligible run time compared to the complete verification run time. We show in Fig. 8a for each task verified with CPA-Seq$^{\Delta}$ and UAutomizer$^{\Delta}$, the CPU time required by the full verification run (x-axis) and the CPU time of that run spent for diffCond plus the reducer (y-axis). The time required by diffCond + reducer does not depend on the run time of the verifier, and it is below 60 s for all tasks.



(a) Full verification, diffCond + reducer

(b) Predicate$^{\circlearrowright}$, Predicate$^{\Delta}$

(c) Predicate$^{\Delta}$, Predicate$^{\Delta\circlearrowright}$

Fig. 8: CPU time (in s) of (a) full difference-verification runs and the time spent for the two diff. components diffCond + reducer, (b) Predicate with precision reuse (Predicate$^{\circlearrowright}$) vs. Predicate with difference verification (Predicate$^{\Delta}$), and (c) Predicate$^{\Delta}$ vs. Predicate$^{\Delta}$ with precision reuse (Predicate$^{\Delta\circlearrowright}$)

Claim 5 (Difference verification with conditions complementary). To show that difference verification with conditions complements existing incremental verification, we need to compare difference verification with conditions against an existing incremental approach. Looking at existing approaches that are (1) available as replication artifact and (b) able to run on verification tasks from sv-benchmarks, we identified two: both based on precision reuse, one implemented in CPAchecker [16] and one in Ultimate [49]. We use the one in CPAchecker. Figure 8b shows the CPU time of precision reuse with Predicate, called Predicate$^{\circlearrowright}$ (x-axis) against our difference verification with Predicate, called Predicate$^{\Delta}$ (y-axis). Many tasks are solved efficiently by both techniques (large cluster in lower left). For the remaining hard tasks, difference verification is often faster than precision reuse, or precision reuse cannot even solve the task (points below the diagonal and on right edge). This shows that difference verification with conditions can improve on precision reuse for a significant number of tasks. It can thus complement existing incremental techniques.

Claim 6 (Combinations sometimes beneficial). We combined difference verification with conditions with precision reuse, called Predicate$^{\Delta\circlearrowright}$. Figure 8c shows that this combination rarely becomes faster than difference verification Predicate$^{\Delta}$ alone. In the worst case, the combination even slows down because precision reuse tracks previously used predicates from the beginning while difference verification would only detect the necessary ones lazily. This more precise abstraction leads to more, sometimes unnecessary computations. Nevertheless, the combination can solve 29 tasks that neither Predicate, its difference verifier, nor precision reuse can solve alone. Thus, while a combination of the two incremental techniques is not beneficial in general, it can be.

5.3 Threats to Validity

External Validity. (1) Our benchmark tasks might not represent real program changes, and thus, our results might not transfer to reality. However, we built our tasks from a well-established collection of software-verification problems, which are considered relevant in the verification community. Moreover, many of the combined programs implement known algorithms (greatest common divisor, Newton approximation of a sine function, Taylor expansion of a square root) or are derived from real applications (OpenSSL, SystemC design, leader election). Also, our combination is not uncommon in practice. Such combination patterns e.g. result from implementing the strategy pattern. Finally, our task set contains pairs of programs whose only difference is a bug fix to eliminate the reachability of the __VERIFIER_error() call. We believe that similar fixes are done in practice to eliminate bugs. (2) We compared our approach only with a single existing approach for incremental verification, and this comparison is restricted to a single verifier. Our observations may not apply to different incremental verification approaches or different verifiers. The same holds for the combination of difference verification with orthogonal, incremental verification approaches. *Internal Validity.* (3) The implementation of the DIFFCOND algorithm may contain bugs, and thus, produces conditions that also exclude modified paths. We would expect that such a bug also excludes error paths. Since we never observed false proofs, we assume this is unlikely. (4) Difference verification with CPA-SEQ and UAUTOMIZER could appear improved simply because we separated verification from the execution of DIFFCOND + REDUCER and granted both runs a limit of 900 s. But the sum of the two times are always below 900 s for all correctly solved tasks.

6 Related Work

Equivalence Checking. Regression verification [27, 28, 55, 56], SYMDIFF [23], UC-Klee [48], and other approaches [4, 26] check whether the input-output behavior of the original and modified method or program is the same. Differential assertion checking [38] inspects whether the original and modified program trigger the same assertions when given the same inputs. Equivalence checking does not need to be restricted to a simple yes or no answer. Semantic Diff [35] reports all dependencies between variables and input values that occur either in the original or modified program. Conditional equivalence [37] infers under which input assumption the original and modified program produce the same output. Over-approximation of the differences between the original and modified program was also investigated [45]. Differential symbolic execution [46] compares function summaries and constructs a delta that describes the input values on which the summaries are unequal. Partition-based regression verification [19] splits the program input space into inputs on which original and modified program behave equivalently and those on which the two programs are unequal. Equivalence checking is not directly tailored to property verification, but determining when the original and modified programs may behave differently is similar to the goal of the DIFFCOND algorithm.

Result Adaption. Incremental data-flow analysis [51], Reviser [3], and IncA [57, 58] adapt the existing data-flow solution to program modifications. Similarly, incremental abstract interpretation [52] adapts the solution of the abstract interpreter. Incremental model checking in the modal-μ calculus [54] adapts a previous fixed point and restarts the fixed-point iteration. Other approaches [18, 20] model data-flow analysis and verification as computation of attributed parse trees. A change results in an update of the attributed parse tree. Extreme model checking [33] reuses valid parts of the abstract reachability graph (ARG) and resumes the state-space exploration from those nodes with invalid successors. Incremental state-space exploration [41] reuses a previous state-space graph to prune the current exploration. HiFrog [1] and eVolCheck [25] implement an approach that reuses function summaries and recomputes invalid summaries [53]. UAutomizer adapts a previous trace abstraction [49], a set of Floyd-Hoare automata that describe infeasible error paths, to reuse it on the modified program. While result adaption uses the same verification technique for original and modified program, our approach may use different techniques.

Reusing Intermediate Results. Green [59], GreenTrie [36], and Recal [2] support the reuse of constraint proofs. Similarly, iSaturn [44] supports the reuse of SAT results of Boolean constraints that are identical. Precision reuse [16] reuses the precision of an abstraction, e.g., which variables or predicates to track, from a previous verification run. These approaches are orthogonal to our approach. In the experiments, we even combined precision reuse [16] with our approach.

Skipping Unaffected Verification Steps. Regression model checking [60] stops exploration of a state as soon as no program change can be reached from that state. Directed incremental [47, 50] and memoized [61] symbolic execution restrict the exploration to paths that may be affected by the program change. Additionally, memoized symbolic execution does not check constraints as long as the path prefix is unchanged. The Dafny verifier rechecks methods affected by a change reusing unchanged verification conditions [42]. iCoq [21, 22] detects and only rechecks those Coq proofs that are affected by a change in the Coq project. These ideas are similar to ours but are tailored to specific techniques.

7 Conclusion

Software is frequently changed during development. Verification techniques must deal with repeatedly verifying nearly the same software again and again. To be able to construct efficient incremental verifiers from off-the-shelf components, we introduce *difference verification with conditions*, which steers an arbitrary existing verifier to reverify only the changed parts. Compared to existing techniques, our approach is tool-agnostic and can be used with arbitrary algorithms for change analysis. We provide an implementation of a change analysis as reusable component, which we combined with three existing verifiers. In a thorough evaluation on more than 10 000 tasks, we showed the effectiveness and efficiency of difference verification with conditions.

Data Availability Statement. diffCond and all our data are available for replication and to construct further difference verifiers on our supplementary web page[9] and in a replication package on Zenodo [12].

[9] https://www.sosy-lab.org/research/difference/

References

1. Alt, L., Asadi, S., Chockler, H., Even-Mendoza, K., Fedyukovich, G., Hyvärinen, A.E.J., Sharygina, N.: HiFrog: SMT-based function summarization for software verification. In: Proc. TACAS. pp. 207–213. LNCS 10206 (2017). https://doi.org/10.1007/978-3-662-54580-5_12
2. Aquino, A., Bianchi, F.A., Chen, M., Denaro, G., Pezzè, M.: Reusing constraint proofs in program analysis. In: Proc. ISSTA. pp. 305–315. ACM (2015). https://doi.org/10.1145/2771783.2771802
3. Arzt, S., Bodden, E.: Reviser: Efficiently updating IDE-/IFDS-based data-flow analyses in response to incremental program changes. In: Proc. ICSE. pp. 288–298. ACM (2014). https://doi.org/10.1145/2568225.2568243
4. Backes, J., Person, S., Rungta, N., Tkachuk, O.: Regression verification using impact summaries. In: Proc. SPIN. pp. 99–116. LNCS 7976, Springer (2013). https://doi.org/10.1007/978-3-642-39176-7_7
5. Beyer, D.: Advances in automatic software verification: SV-COMP 2020. In: Proc. TACAS (2). pp. 347–367. LNCS 12079, Springer (2020). https://doi.org/10.1007/978-3-030-45237-7_21
6. Beyer, D., Dangl, M.: Strategy selection for software verification based on boolean features: A simple but effective approach. In: Proc. ISoLA. pp. 144–159. LNCS 11245, Springer (2018). https://doi.org/10.1007/978-3-030-03421-4_11
7. Beyer, D., Dangl, M., Wendler, P.: Boosting k-induction with continuously-refined invariants. In: Proc. CAV. pp. 622–640. LNCS 9206, Springer (2015). https://doi.org/10.1007/978-3-319-21690-4_42
8. Beyer, D., Gulwani, S., Schmidt, D.: Combining model checking and data-flow analysis. In: Handbook of Model Checking, pp. 493–540. Springer (2018). https://doi.org/10.1007/978-3-319-10575-8_16
9. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The software model checker Blast. Int. J. Softw. Tools Technol. Transfer **9**(5-6), 505–525 (2007). https://doi.org/10.1007/s10009-007-0044-z
10. Beyer, D., Henzinger, T.A., Keremoglu, M.E., Wendler, P.: Conditional model checking: A technique to pass information between verifiers. In: Proc. FSE. ACM (2012). https://doi.org/10.1145/2393596.2393664
11. Beyer, D., Henzinger, T.A., Théoduloz, G.: Configurable software verification: Concretizing the convergence of model checking and program analysis. In: Proc. CAV. pp. 504–518. LNCS 4590, Springer (2007). https://doi.org/10.1007/978-3-540-73368-3_51
12. Beyer, D., Jakobs, M.C., Lemberger, T.: Replication package for article 'Difference verification with conditions'. Zenodo (2020). https://doi.org/10.5281/zenodo.3954933
13. Beyer, D., Jakobs, M.C., Lemberger, T., Wehrheim, H.: Reducer-based construction of conditional verifiers. In: Proc. ICSE. pp. 1182–1193. ACM (2018). https://doi.org/10.1145/3180155.3180259
14. Beyer, D., Keremoglu, M.E.: CPAchecker: A tool for configurable software verification. In: Proc. CAV. pp. 184–190. LNCS 6806, Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_16
15. Beyer, D., Keremoglu, M.E., Wendler, P.: Predicate abstraction with adjustable-block encoding. In: Proc. FMCAD. pp. 189–197. FMCAD (2010)
16. Beyer, D., Löwe, S., Novikov, E., Stahlbauer, A., Wendler, P.: Precision reuse for efficient regression verification. In: Proc. FSE. pp. 389–399. ACM (2013). https://doi.org/10.1145/2491411.2491429

17. Beyer, D., Löwe, S., Wendler, P.: Benchmarking and resource measurement. In: Proc. SPIN. pp. 160–178. LNCS 9232, Springer (2015). https://doi.org/10.1007/978-3-319-23404-5_12
18. Bianculli, D., Filieri, A., Ghezzi, C., Mandrioli, D.: Syntactic-semantic incrementality for agile verification. SCICO **97**, 47–54 (2015). https://doi.org/10.1016/j.scico.2013.11.026
19. Böhme, M., d. S. Oliveira, B.C., Roychoudhury, A.: Partition-based regression verification. In: Proc. ICSE. pp. 302–311. IEEE (2013). https://doi.org/10.1109/ICSE.2013.6606576
20. Carroll, M.D., Ryder, B.G.: Incremental data-flow analysis via dominator and attribute updates. In: Proc. POPL. pp. 274–284. ACM (1988). https://doi.org/10.1145/73560.73584
21. Çelik, A., Palmskog, K., Gligoric, M.: iCoq: Regression proof selection for large-scale verification projects. In: Proc. ASE. pp. 171–182. IEEE (2017). https://doi.org/10.1109/ASE.2017.8115630
22. Çelik, A., Palmskog, K., Gligoric, M.: A regression proof selection tool for Coq. In: Proc. ICSE (Companion Volume). pp. 117–120. ACM (2018). https://doi.org/10.1145/3183440.3183493
23. Chaki, S., Gurfinkel, A., Strichman, O.: Regression verification for multi-threaded programs (with extensions to locks and dynamic thread creation). FMSD **47**(3), 287–301 (2015). https://doi.org/10.1007/s10703-015-0237-0
24. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. J. ACM **50**(5), 752–794 (2003). https://doi.org/10.1145/876638.876643
25. Fedyukovich, G., Sery, O., Sharygina, N.: eVolCheck: Incremental upgrade checker for C. In: Proc. TACAS. pp. 292–307. LNCS 7795, Springer (2013). https://doi.org/10.1007/978-3-642-36742-7_21
26. Felsing, D., Grebing, S., Klebanov, V., Rümmer, P., Ulbrich, M.: Automating regression verification. In: Proc. ASE. pp. 349–360. ACM (2014). https://doi.org/10.1145/2642937.2642987
27. Godlin, B., Strichman, O.: Regression verification. In: Proc. DAC. pp. 466–471. ACM (2009). https://doi.org/10.1145/1629911.1630034
28. Godlin, B., Strichman, O.: Regression verification: Proving the equivalence of similar programs. Software Testing, Verification, and Reliability **23**(3), 241–258 (2013). https://doi.org/10.1002/stvr.1472
29. Heizmann, M., Chen, Y.F., Dietsch, D., Greitschus, M., Hoenicke, J., Li, Y., Nutz, A., Musa, B., Schilling, C., Schindler, T., Podelski, A.: Ultimate Automizer and the search for perfect interpolants (competition contribution). In: Proc. TACAS (2). pp. 447–451. LNCS 10806, Springer (2018). https://doi.org/10.1007/978-3-319-89963-3_30
30. Heizmann, M., Hoenicke, J., Podelski, A.: Refinement of trace abstraction. In: Proc. SAS. pp. 69–85. LNCS 5673, Springer (2009). https://doi.org/10.1007/978-3-642-03237-0_7
31. Heizmann, M., Hoenicke, J., Podelski, A.: Software model checking for people who love automata. In: Proc. CAV. pp. 36–52. LNCS 8044, Springer (2013). https://doi.org/10.1007/978-3-642-39799-8_2
32. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: Proc. POPL. pp. 232–244. ACM (2004). https://doi.org/10.1145/964001.964021
33. Henzinger, T.A., Jhala, R., Majumdar, R., Sanvido, M.A.A.: Extreme model checking. In: Verification: Theory and Practice. pp. 332–358 (2003). https://doi.org/10.1007/978-3-540-39910-0_16

34. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: Proc. POPL. pp. 58–70. ACM (2002). https://doi.org/10.1145/503272.503279
35. Jackson, D., Ladd, D.A.: Semantic Diff: A tool for summarizing the effects of modifications. In: Proc. ICSM. pp. 243–252. IEEE (1994). https://doi.org/10.1109/ICSM.1994.336770
36. Jia, X., Ghezzi, C., Ying, S.: Enhancing reuse of constraint solutions to improve symbolic execution. In: Proc. ISSTA. pp. 177–187. ACM (2015). https://doi.org/10.1145/2771783.2771806
37. Kawaguchi, M., Lahiri, S., Rebelo, H.: Conditional equivalence. Tech. rep., Microsoft Research (2010)
38. Lahiri, S.K., McMillan, K.L., Sharma, R., Hawblitzel, C.: Differential assertion checking. In: Proc. FSE. pp. 345–355. ACM (2013). https://doi.org/10.1145/2491411.2491452
39. Lahiri, S.K., Murawski, A., Strichman, O., Ulbrich, M.: Program Equivalence (Dagstuhl Seminar 18151). Dagstuhl Reports **8**(4), 1–19 (2018). https://doi.org/10.4230/DagRep.8.4.1,
40. Lahiri, S.K., Vaswani, K., Hoare, C.A.R.: Differential static analysis: Opportunities, applications, and challenges. In: Proc. FoSER. pp. 201–204. ACM (2010). https://doi.org/10.1145/1882362.1882405
41. Lauterburg, S., Sobeih, A., Marinov, D., Viswanathan, M.: Incremental state-space exploration for programs with dynamically allocated data. In: Proc. ICSE. pp. 291–300. ACM (2008). https://doi.org/10.1145/1368088.1368128
42. Leino, K.R.M., Wüstholz, V.: Fine-grained caching of verification results. In: Proc. CAV. pp. 380–397. LNCS 9206, Springer (2015). https://doi.org/10.1007/978-3-319-21690-4_22
43. McMillan, K.L.: Interpolation and SAT-based model checking. In: Proc. CAV. pp. 1–13. LNCS 2725, Springer (2003). https://doi.org/10.1007/978-3-540-45069-6_1
44. Mudduluru, R., Ramanathan, M.K.: Efficient incremental static analysis using path abstraction. In: Proc. FASE. pp. 125–139. LNCS 8411, Springer (2014). https://doi.org/10.1007/978-3-642-54804-8_9
45. Partush, N., Yahav, E.: Abstract semantic differencing for numerical programs. In: Proc. SAS. pp. 238–258. LNCS 7935, Springer (2013). https://doi.org/10.1007/978-3-642-38856-9_14
46. Person, S., Dwyer, M.B., Elbaum, S.G., Păsăreanu, C.S.: Differential symbolic execution. In: Proc. FSE. pp. 226–237. ACM (2008). https://doi.org/10.1145/1453101.1453131
47. Person, S., Yang, G., Rungta, N., Khurshid, S.: Directed incremental symbolic execution. In: Proc. PLDI. pp. 504–515. ACM (2011). https://doi.org/10.1145/1993498.1993558
48. Ramos, D.A., Engler, D.R.: Practical, low-effort equivalence verification of real code. In: Proc. CAV. pp. 669–685. LNCS 6806, Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_55
49. Rothenberg, B., Dietsch, D., Heizmann, M.: Incremental verification using trace abstraction. In: Proc. SAS. pp. 364–382. LNCS 11002, Springer (2018). https://doi.org/10.1007/978-3-319-99725-4_22
50. Rungta, N., Person, S., Branchaud, J.: A change impact analysis to characterize evolving program behaviors. In: Proc. ICSM. pp. 109–118. IEEE (2012). https://doi.org/10.1109/ICSM.2012.6405261
51. Ryder, B.G.: Incremental data-flow analysis. In: Proc. POPL. pp. 167–176. ACM (1983). https://doi.org/10.1145/567067.567084

52. Seidl, H., Erhard, J., Vogler, R.: Incremental abstract interpretation. In: From Lambda Calculus to Cybersecurity Through Program Analysis - Essays Dedicated to Chris Hankin on the Occasion of His Retirement. pp. 132–148. LNCS 12065, Springer (2020). https://doi.org/10.1007/978-3-030-41103-9_5
53. Sery, O., Fedyukovich, G., Sharygina, N.: Incremental upgrade checking by means of interpolation-based function summaries. In: Proc. FMCAD. pp. 114–121. FMCAD Inc. (2012)
54. Sokolsky, O.V., Smolka, S.A.: Incremental model checking in the modal mu-calculus. In: Proc. CAV. pp. 351–363. LNCS 818, Springer (1994). https://doi.org/10.1007/3-540-58179-0_67
55. Strichman, O., Godlin, B.: Regression verification — a practical way to verify programs. In: Proc. VSTTE. pp. 496–501. LNCS 4171, Springer (2008). https://doi.org/10.1007/978-3-540-69149-5_54
56. Strichman, O., Veitsman, M.: Regression verification for unbalanced recursive functions. In: Proc. FM. pp. 645–658. LNCS 9995 (2016). https://doi.org/10.1007/978-3-319-48989-6_39
57. Szabó, T., Bergmann, G., Erdweg, S., Voelter, M.: Incrementalizing lattice-based program analyses in Datalog. PACMPL **2**(OOPSLA), 139:1–139:29 (2018). https://doi.org/10.1145/3276509
58. Szabó, T., Erdweg, S., Voelter, M.: IncA: A DSL for the definition of incremental program analyses. In: Proc. ASE. pp. 320–331. ACM (2016). https://doi.org/10.1145/2970276.2970298
59. Visser, W., Geldenhuys, J., Dwyer, M.B.: Green: Reducing, reusing, and recycling constraints in program analysis. In: Proc. FSE. pp. 58:1–58:11. ACM (2012). https://doi.org/10.1145/2393596.2393665
60. Yang, G., Dwyer, M.B., Rothermel, G.: Regression model checking. In: Proc. ICSM. pp. 115–124. IEEE (2009). https://doi.org/10.1109/ICSM.2009.5306334
61. Yang, G., Păsăreanu, C.S., Khurshid, S.: Memoized symbolic execution. In: Proc. ISSTA. pp. 144–154. ACM (2012). https://doi.org/10.1145/04000800.2336771
62. Yoo, S., Harman, M.: Regression testing minimization, selection, and prioritization: A survey. STVR **22**(2), 67–120 (2012). https://doi.org/10.1002/stv.430