Conc-iSE: Incremental Symbolic Execution of Concurrent Software

Shengjian Guo, Markus Kusano
Department of ECE
Virginia Tech
Blacksburg, VA, USA

Chao Wang
Department of CS
University of Southern California
Los Angeles, CA, USA

ABSTRACT

Software updates often introduce new bugs to existing code bases. Prior regression testing tools focus mainly on test case selection and prioritization whereas symbolic execution tools only handle code changes in sequential software. In this paper, we propose the first incremental symbolic execution method for concurrent software to generate new tests by exploring only the executions affected by code changes between two program versions. Specifically, we develop an inter-thread and inter-procedural change-impact analysis to check if a statement is affected by the changes and then leverage the information to choose executions that need to be re-explored. We also check if execution summaries computed in the previous program can be used to avoid redundant explorations in the new program. We have implemented our method in an incremental symbolic execution tool called *Conc-iSE* and evaluated it on a large set of multithreaded C programs. Our experiments show that the new method can significantly reduce the overall symbolic execution time when compared with state-of-the-art symbolic execution tools such as KLEE.

CCS Concepts

•**Software and its engineering** → *Software verification and validation;* Software testing and debugging; Software evolution;

Keywords

Symbolic execution, Concurrency, Partial order reduction, Weakest precondition

1. INTRODUCTION

As software evolves, updates made from the addition of new features or patches may introduce new bugs. While some regression testing tools can leverage code changes between two software versions to reduce the testing cost, they focus primarily on selection and test case prioritization as opposed to the creation of new test cases. In contrast, symbolic execution is a technique for automatically generating new tests, and, more recently [30, 32, 46], has been used in regression testing to reduce the overall cost for sequential software testing. Specifically, prior work uses a conservative static analysis to estimate the impact of the code changes and then leverage the information to avoid re-executing program paths that are not affected by these code changes. However, these methods only handle code changes in sequential software. Furthermore, they rely on an overly conservative analysis to estimate the change impact, without making use of the more accurate information available from previous symbolic execution runs.

ASE'16, September 3–7, 2016, Singapore, Singapore

http://dx.doi.org/10.1145/2970276.2970332

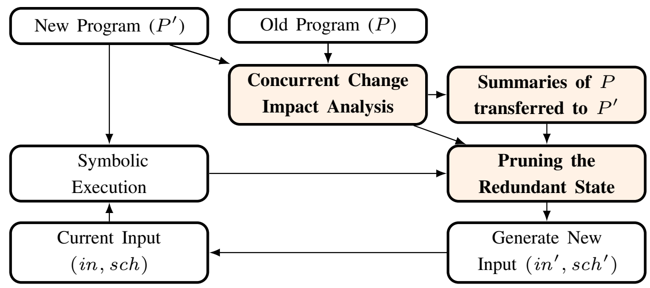

Figure 1: Summary-based incremental symbolic execution.

In this paper, we propose *Conc-iSE*: an incremental symbolic execution method for concurrent programs. Figure 1 shows the overall flow of our new method. We take old (P) and new (P') program versions, together with a set of execution summaries of P, as input and iteratively explore new execution paths through P'. As we will show, we use supplementary information from P (the execution summaries) as well as code changes between P and P' to perform the incremental analysis.

The standard and non-incremental symbolic execution procedure is shown in the lower half of Figure 1, which starts from an arbitrary initial test (in, sch) of P' and repeatedly generates new tests for P'. Here, in denotes the data input and sch denotes the thread interleaving schedule. We assume P' is a deterministic program whose execution is completely decided by the pair (in, sch). During symbolic execution, new states are generated to explore alternate branches and alternate thread interleaving schedules. For each new state, the symbolic execution engine generates a new pair (in', sch') containing the data input and thread schedule to reach the new state. In the non-incremental approach, no information about previously explored executions in P and code changes made to P' are used to determine if a state is redundant: program executions equivalent to behavior in P are re-explored in P'.

Incremental symbolic execution, in contrast, considers two program versions P and P' while assuming P is a prior version that has already been explored symbolically. The goal is to explore only the *new* behavior in P'. Prior works on incremental symbolic execution for sequential programs [30, 32, 46] used a forward change-impact analysis, built on the idea of program slicing [43], to determine if a statement in P' was affected by a modification; only affected portions of the code in P' were explored again during symbolic execution. Our first insight is that performing a change-impact analysis using a conservative static analysis alone often results in the testing of redundant executions. This is because a conservative static analysis, such as program slicing, ignores the actual values of variables in the program. As we will show in Section 2, even if a statement is modified (from P to P'), it may be that paths affected by this modification are equivalent to some paths in the previous version. To define a more accurate equivalence class of execution paths, we make use of the execution summaries from P while testing P', as opposed to performing only a conservative change-impact analysis. At a high level, the execution summaries, defined at each global control state, capture the set of all explored executions starting from s. The summaries are computed backwardly using a weakest-precondition computation.

We also propose an inter-thread and inter-procedural change impact analysis for handling both sequential and concurrent programs. It consists of a forward analysis and a backward analysis. The forward change-impact analysis computes the set of statements that *may be affected* by code changes from P to P'; this is used to avoid executing portions of P' unaffected by statements that are changed from P to P'. The backward change-impact analysis computes the set of statements that *may affect* statements that are changed from P to P'; this is used to determine if an execution summary from the old version P can be carried over to the new version P'. Intuitively, in both cases, if a code modification in P' only affects a small number of statements, then much of P' is the same as P.

The combination of execution summaries and change-impact analysis, as well as their interaction with the baseline symbolic execution procedure, is shown in Figure 1. Recall that prior incremental symbolic execution techniques [30, 32, 46] only handled sequential programs, whereas *Conc-iSE* is the first incremental symbolic execution algorithm capable of handling concurrent programs. Specifically, when a new state in P' is generated, we check both the change-impact information and the execution summaries to see if the state is in the unmodified section of the program, or if it is equivalent to some previously explored execution in P. If either condition is true, then the new state is redundant and can be skipped.

Conc-iSE differs from the prior works on regression testing of multithreaded programs [17, 49, 38]. In Jagannath et al. [17] and Yu et al. [49], for example, the primary focus was on test case selection and test case prioritization, i.e., to detect certain concurrency bugs quicker by heuristically selecting test cases and scheduling them in certain orders, as opposed to generating new test cases. In contrast, our method focuses on making symbolic execution *incremental*, which will benefit test case generation. Our method also differs from the work by Terragni et al. [38], which symbolically analyzes the alternative interleavings of some concrete executions based on the trace logs. Unlike our method, it does not perform symbolic execution based test input generation to explore both intra-thread program paths and inter-thread interleavings.

We have implemented our method in a software tool using LLVM [23] and Cloud9 [5]. We used LLVM to implement our forward and backward change-impact analysis algorithms, and used the KLEE symbolic virtual machine in Cloud9 as the baseline to implement our incremental symbolic execution algorithm. We also extended KLEE to robustly handle POSIX thread routines and implement the state-of-the-art dynamic partial-order reduction (DPOR) technique [9]. We evaluated *Conc-iSE* on a large set of multithreaded C programs, including benchmarks from the Software Verification Competition [37] and real-world applications that are open-source implementations of non-blocking data structures [29]. In total, our benchmarks contain 14 programs, with a total of 70 different versions and 34,926 lines of code. Empirically, we showed our method can significantly reduce the overall testing time when compared with state-of-the-art symbolic execution techniques.

To sum up, this paper makes the following contributions:

```
x = 15, y = 5;
```

```
/**  [Thread 1]  **/
1:  a = x;
2:  y = 10;
3:
/**  [Thread 2]  **/
4:  x = 10;
5:  b = y;
6:
```

```
/**  [Thread 1]  **/
1:  a = x;
2:  y = 10;
3:
/**  [Thread 2]  **/
4:  x = 5; //modified
5:  b = y;
6:
```

```
assert(a>=10);
assert(b>=5);
```

Figure 2: Example program: old (left) and new (right) versions.

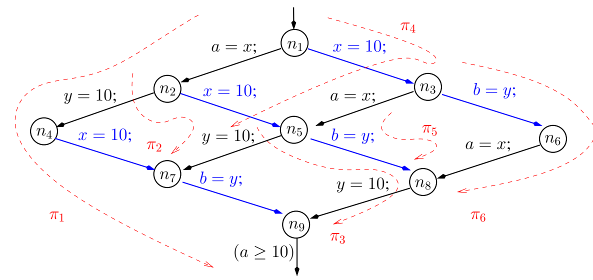

Figure 3: Interleaved executions of old version: $\pi_1, \ldots, \pi_6$.

- We propose an incremental symbolic execution algorithm capable of handling code changes in both sequential and concurrent programs.
- We develop a new execution summary-based algorithm for pruning away redundant paths and thread interleavings during incremental symbolic execution.
- We implement our new method in a software tool and evaluate it on a large set of benchmarks to demonstrate its effectiveness at decreasing regression testing time.

2. MOTIVATING EXAMPLES

In this section, we illustrate the main ideas behind our new method.

2.1 Pruning with Change-Impact Analysis

Consider the example in Figure 2. The old program on the left-hand side has two threads accessing the shared variables x and y. They are initialized to 15 and 5, respectively. After executing both threads, the two assertions are checked. The new program is shown on the right-hand side; the only modification between the two programs is on Line 4, where `x=10` is changed to `x=5`. First, note that although the modification is in the second thread, due to the sharing of variable x, Line 1 in the first thread is also affected. Such impacted instructions cannot be identified by existing algorithms [30, 32, 46] since they were not designed for analyzing concurrent programs; our new change-impact analysis solves this problem.

Second, there are six possible executions of the old program, as shown in the abstract state transition graphs in Figure 3. State-of-the-art partial-order reduction (POR) techniques [11, 9, 20] can reduce the number of executions to four. Our method does even better by reducing the number of executions to two. Specifically, in Figure 3 each node denotes a global control state, e.g., $n_1 = (1, 4)$ means Thread 1 is at Line 1 and Thread 2 is at Line 4, while $n_2 = (2, 4)$ means they are at Lines 2 and 4. After POR, only four executions remain as shown in the left-hand-side execution tree in Figure 4. The reason why π_2 and π_6 are skipped is because they are equivalent to π_1 and π_5, respectively. That is, executing the two independent instructions $y = 10$ and $x = 10$ in different orders lead to the same result.

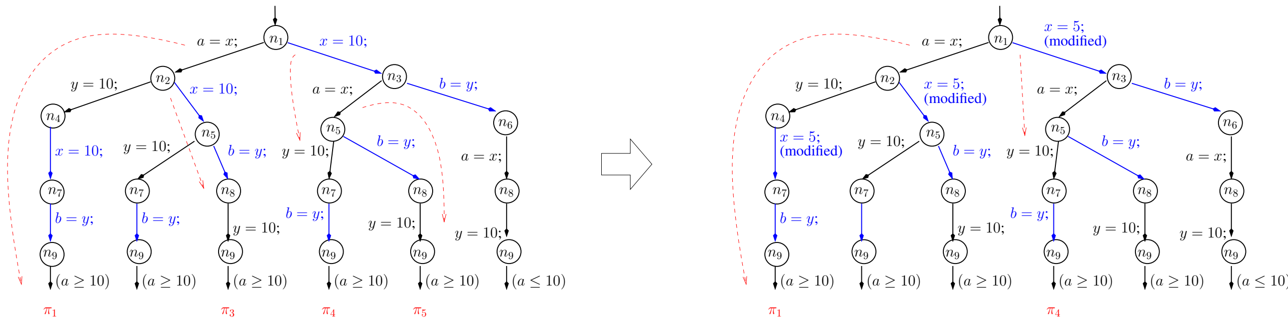

Figure 4: Executions explored by incremental symbolic execution in the old program (left) and the new program (right).

By leveraging the concurrent change-impact analysis, our method can identify even more redundant executions than POR. Specifically, since the code change on Line 4 does not impact Line 2 or Line 5 or *assert(b>=5)*, we do not need to re-explore the different execution orders of $y = 10$ and $b = y$. Because of this reason, as shown in the right-hand-side tree in Figure 4, our method can reduce the four executions to two (π_1 and π_4).

In this work, we assume that assertions are embedded in the individual threads. As such, the assertion conditions always refer to local variables, or local copies of global variables, which is consistent with the assumptions made in prior works on POR [11, 9, 20]. It is worth pointing out that, in this example, the assertion conditions are also important: if the assertion were *assert(a>=b)*, then it is no longer safe to skip π_3 and π_5. Details of our new change-impact analysis algorithm and its application to incremental symbolic execution are presented in Section 5.

2.2 Pruning with Execution Summary

In addition to leveraging the forward change-impact analysis, we also propose an orthogonal pruning technique based on a backward change-impact analysis. That is, instead of computing the set of instructions that *may be affected by the changed instructions*, we compute the set of instructions that *may affect the changed instructions*. Details of the backward change-impact analysis and its application are presented in Section 6. Here, we briefly illustrate the main ideas using an example.

Consider the two versions of a sequential program in Figure 5, where the old version is on the left, and the new version is on the right. The only modification is on Line 1; the condition is changed from $(x > 0)$ to $(x \geq 0)$. From the forward change-impact analysis described in Section 2.1, or for that matter, existing methods for incremental symbolic execution [30, 32, 46], we know that all the other lines in the new program are affected by the change. Therefore, it seems that no redundant executions can be pruned away.

However, if we divide the initial program state into three subsets, denoted $(x > 0)$, $(x = 0)$, and $(x < 0)$, respectively, then it is clear that only when $(x = 0)$, the modified program behave different from the original program. In the old version, such case was handled by paths π_3 and π_4, but in the new version, it is handled by paths π_1 and π_2. Therefore, instead of re-exploring all four paths, we only need to re-explore π_1 and π_2.

The question then is how to figure out, algorithmically, that paths π_3 and π_4 are indeed redundant. Our solution in *Conc-iSE* is to compute, for each global control state s, a summary of all the explored executions starting from s in the old program version. For example, the summary at n_4, with respect to *assert(b≠0)*, would be $\mathsf{PS}[n_4] = (y > 0) \wedge (x \neq 1) \vee (y \leq 0) \wedge (x \neq 3)$. This summary is created from the union of the weakest precondition of $(b \neq 0)$ along the two outgoing paths.

```
1:  if (x>0)
2:        a = x+2;
3:  else
4:        a = x-2;
5:  if (y>0)
6:        b = a+1;
7:  else
8:        b = a-1;
9:  assert(b!=0);
```

```
1:  if (x>=0) //modified
2:        a = x+2;
3:  else
4:        a = x-2;
5:  if (y>0)
6:        b = a+1;
7:  else
8:        b = a-1;
9:  assert(b!=0);
```

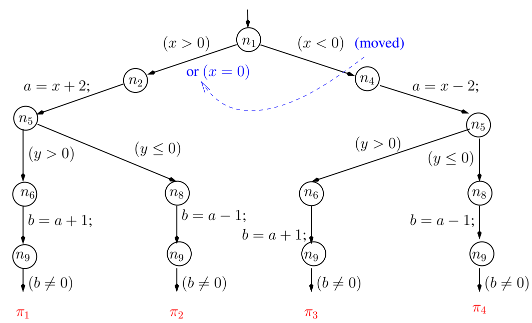

Figure 5: Although all instructions are impacted by the code change on Line 1, not all four paths need to be re-explored.

Since the code changes on Line 1 does not affect the aforementioned weakest precondition computation, the summary can be carried over to the new program. During the analysis of the new program, we can stop an execution as soon as the path condition, denoted $pcon[n_4] = (x < 0)$, falls within the set $\mathsf{PS}[n_4]$ of explored executions. This early termination is safe because if $pcon[n_4] \wedge \neg\mathsf{PS}[n_4]$ is unsatisfiable, re-exploring the executions starting from n_4 would not lead to any new error.

3. PRELIMINARIES

In this section, we establish the notation and review our baseline symbolic execution algorithm for multithreaded programs.

3.1 Multithreaded Programs

We assume each program P consists of a finite set of threads, $\{T_1, \ldots, T_m\}$, and a set $SVar$ of shared variables. Each thread T_i, where $1 \leq i \leq m$, has a set $LVar_i$ of local variables. Instructions from different threads are executed in an interleaved fashion. Each time an instruction st is executed, it produces an event $e = \langle tid, st, l, l' \rangle$, where tid is the thread id, while l and l' are the program locations before and after executing st. If there are multiple execution instances of st, each instance is represented by a different event.

A concrete state of the program P consists of the program location l_i of every thread T_i, where $1 \leq i \leq m$, and the values of all variables in $SVar$ and $LVar_i$. In contrast, the abstract state, or the so-called *global control state (GCS)* $s = \langle l_1, \ldots, l_m \rangle$, consists of the program locations only. In other words, each GCS represents the set of all concrete states that share the same program locations but have potentially different values of the program variables.

Let v_l and $cond_l$ be the thread-local variables and conditions, while v_g and $cond_g$ be the shared (global) variables and conditions, respectively. Depending on whether an event accesses shared variables, we classify it into one of the following categories:

- α-operation: a local assignment $v_l := exp_l$;
- β-operation: a local branch `assume(`$cond_l$`)`;
- γ-operation: a global operation defined as either
  - a global write $v_g := exp_l$ or read $v_l := v_g$; or
  - a thread synchronization operation.

Given a program P, the set of all possible executions is captured by a *generalized interleaving graph (GIG)* [12], where nodes are global control states and edges are events. The root node corresponds to the program's initial state. Leaf nodes correspond to the end of normal/faulty executions. Each internal node may have one outgoing edge corresponding to an α-operation, k outgoing edges corresponding to β-operations, or k outgoing edges where $k \geq 2$ is the number of enabled γ-operations from different threads.

We make a distinction between thread-local operations and global operations since they have different impacts during symbolic execution. Global operations (γ) directly affect the thread interleaving order, while β-operations directly affect the path taken by each thread. In contrast, α-operations do not directly affect the selection of any program path or thread interleaving.

Without loss of generality, we assume all conditional expressions use local variables or local copies of global variables [11]. The execution of an `if(c)-else` statement, for example, can be represented by `assume(`c`)` if we take the *then*-branch, and `assume(`$\neg c$`)` if we take the *else*-branch. Properties of interest are represented by assertions of the form `assert(c)`, which means `if(!c)abort`. Therefore, we can use the special event **abort** to denote faulty program termination and **halt** to denote normal program termination.

3.2 Baseline Symbolic Execution

Following the majority of prior works on symbolic execution, we assume that the program under test is terminating and thus each execution has a finite length [3]. We also assume the program is deterministic, i.e., the sequence of instructions will be completely determined by (in, sch), where in is the data input and sch is the thread schedule. Therefore, (in, sch) implicitly represents a concrete execution of a program. In contrast, $\pi = (*, sch)$ represents a symbolic execution where $*$ is the symbolic data input and $sch = e_1 \ldots e_n$ is an order of the executed events.

Algorithm 1 shows the baseline procedure for concurrent programs, which follows prior works such as [33, 5, 12]. Initially, EXPLORE is invoked with the symbolic initial state s_0. Then, depending on the type of the current state s, we either explore a thread-local branch or schedule a context switch. A *pivot point* is a GIG node with multiple outgoing edges. A node corresponding to β-operation is called a *branching pivot point* (b-PP); a node corresponding to γ-operation is called an *interleaving pivot point* (i-PP). Specifically, if s is an *i-PP* node, we recursively explore the next γ event from each thread; if s is a *b-PP* node, we recursively explore the next thread-local branch; and if s is a non-branching node, we explore the unique next event. Upon reaching a leaf node the current execution ends. At this point, the procedure pops the current state s from the stack S before returning from EXPLORE(s).

During backtracking, we always stop at the last unexplored pivot point (i-PP or b-PP) and try to flip a previous decision to compute a new execution. By flipping a previous decision at an i-PP node, we get (in, sch'), where sch' is a new thread schedule. By flipping a previous decision at a b-PP node, we get (in', sch), where in' is a new data input. In both cases, the new execution will be the same as the previous one up to the pivot point. After the pivot point, however, it will be an uncontrolled execution.

Algorithm 1 Baseline Symbolic Execution.

```
Initially: Stack S = {s0}; run EXPLORE(s0) with the symbolic initial state s0.
EXPLORE(s)
    S.push(s);
    if (s is an i-PP node)
        while (∃t ∈ (s.enabled \ s.done))
            s' ← NEXTSTATE(s, t);
            EXPLORE(s');
            s.done ← s.done ∪ {t};
    else if (s is a b-PP node)
        while (∃t ∈ (s.branch \ s.done))
            s' ← NEXTSTATE(s, t);
            EXPLORE(s');
            s.done ← s.done ∪ {t};
    else if (s is an internal node)
        s' ← NEXTSTATE(s, t);
        EXPLORE(s');
    else
        //end of execution – do nothing;
    S.pop();

NEXTSTATE(s, t)
    let s = ⟨pcon, M, enabled, branch, done⟩;
    if ( t is halt )
        s' ← normal_end_state;
    else if ( t is abort )
        s' ← faulty_end_state;
    else if ( t is assignment v := exp )
        s' ← ⟨pcon, M[v ↦ exp]⟩;
    else if ( t is assume(c) and M[pcon ∧ c] is satisfiable )
        s' ← ⟨pcon ∧ c, M⟩;
    else
        s' ← infeasible_state;
    return s';
```

We assume that each symbolic program state $s \in S$ is a tuple $\langle pcon, \mathcal{M}, enabled, branch, done \rangle$, where $pcon$ is the path condition from s_0 to s, $\mathcal{M}$ is the memory map, $enabled$ is the set of γ-events when s is an i-PP node, $branch$ is the set of β-events when s is a b-PP node, and $done$ is the set of explored (β or γ) events.

The initial state s_0 is $\langle true, \mathcal{M}_{init}, \ldots \rangle$, where $true$ means the state is always reachable, and $\mathcal{M}_{init}$ is the initial memory map. Each instruction (t) is executed by NEXTSTATE(s,t) as follows:

- If t is **halt**, the current execution ends without error.
- If t is **abort**, we have detected an error.
- If t is an assignment *v:=exp*, we update the memory map $\mathcal{M}$ by changing the content of v to exp.
- If t is *assume(c)*, we set the path condition to $(pcon \wedge c)$.

4. THE INCREMENTAL SYMBOLIC EXECUTION ALGORITHM

Our incremental procedure, shown in Algorithm 2, has two significant differences from the baseline procedure in Algorithm 1. For brevity, we only highlight the parts that are different.

First, the input has changed. Instead of taking one program as input, we take both the old and the new programs (P and P'). Prior to our symbolic execution of the new program P', we compute the forward impacted set $\mathsf{IS}_{\mathsf{fwd}}$ and the backward impacted set $\mathsf{IS}_{\mathsf{bwd}}$. In addition, we transfer the table PS of execution summaries computed in P to the new program P'. For each state s, the set of explored executions starting from s is denoted $\mathsf{PS}[s]$.

Second, we add Lines 27–29 and 32–34 inside NEXTSTATE. They leverage $\mathsf{IS}_{\mathsf{fwd}}$, $\mathsf{IS}_{\mathsf{bwd}}$, and $\mathsf{PS}[s]$ to decide, at each symbolic execution step ($s \xrightarrow{t} s'$), if all executions starting at the next state s'

are redundant. Specifically, if $t.inst \notin \mathsf{IS_{fwd}}$, the current branching statement is not in the impacted set. Since which branch to execute at s is immaterial, if one of the branches has previously been explored, we can force an early termination of the current execution.

Similarly, if $t.inst \notin \mathsf{IS_{bwd}}$, the weakest precondition computation, upon which the execution summary is computed, would not be affected by the code changes. Therefore, we can carry the summary $\mathsf{PS}[s]$ from P to P'. If the current path condition $pcon$, in the modified program, is subsumed by $\mathsf{PS}[s]$ then continuing the execution from s would lead to no new errors. In such case, we can force an early termination of the current execution.

Algorithm 2 Incremental Symbolic Execution.

```
IS_fwd ← ComputeForwardImpactedSet(P, P');
IS_bwd ← ComputeBackwardImpactedSet(P, P');
PS[s] ← the summary at s computed in previous program P;
...
NextState(s, t)
    let s = ⟨pcon, M, enabled, branch, done⟩;
    if ( t is halt )
        s' ← normal_end_state;
    else if ( t is abort )
        s' ← faulty_end_state;
    else if ( t is assignment v := exp )
        if ( t.inst ∉ IS_bwd and pcon ⟹ PS[s] )
            s' ← early_termination_state;
        else
            s' ← ⟨pcon, M[v ↦ exp]⟩;
    else if ( t is assume(c) and M[pcon ∧ c] is satisfiable )
        if ( t.inst ∉ IS_fwd and another branch has been explored )
            s' ← early_termination_state;
        else
            s' ← ⟨pcon ∧ c, M⟩;
    else
        s' ← infeasible_state;
    return s';
```

Example. For the program in Figure 5, the code changes on Line 1 would only invalidate the summary $\mathsf{PS}[n_1]$. Therefore, although we cannot force an early termination at n_1, we can leverage the summary at other nodes to prune away redundant executions. In particular, when the execution reaches either n_2 or n_4, we can terminate the execution immediately. This is because both $pcon[n_2] \wedge \neg\mathsf{PS}[n_2]$ and $pcon[n_4] \wedge \neg\mathsf{PS}[n_4]$ are unsatisfiable. Specifically,

$$\mathsf{PS}[n_2] = (y > 0) \wedge (x \neq -3) \vee (y \leq 0) \wedge (x \neq -1)$$
$$\mathsf{PS}[n_4] = (y > 0) \wedge (x \neq 1) \vee (y \leq 0) \wedge (x \neq 3)$$

Furthermore, $pcon[n_2] = (x \geq 0)$, and $pcon[n_4] = (x < 0)$. Therefore, we can check $pcon[n_2] \wedge \neg\mathsf{PS}[n_2]$ as follows:

$$= (x \geq 0) \wedge ((y \leq 0) \vee (x = -3)) \wedge ((y > 0) \vee (x = -1))$$
$$= \mathsf{false}$$

We can also check $pcon[n_4] \wedge \neg\mathsf{PS}[n_4]$ as follows:

$$= (x < 0) \wedge ((y \leq 0) \vee (x = 1)) \wedge ((y > 0) \vee (x = 3))$$
$$= \mathsf{false}$$

The above checks indicate that no new errors can be detected by continuing from n_2 and n_4. Therefore, we terminate the symbolic execution immediately without exploring the remaining paths.

In the remainder of this paper, we will present our algorithms for conducting the forward and backward change-impact analysis, as well as the redundancy pruning based on execution summaries.

5. CHANGE-IMPACT ANALYSIS

The first important component of our incremental analysis is the detection and characterization of code changes, called the change-impact analysis (CIA) [24]. The identification of code changes requires comparison of two program versions by matching their representations, often in the form of flow graphs [31], tree representations [47], or locations in source files.

5.1 Computing the Impacted Sets

Our new change-impact analysis for concurrent programs takes two program versions P and P' as input and returns two impacted sets. One impacted set is $\mathsf{IS_{fwd}}$, the forwardly impacted set, while the other impacted set is $\mathsf{IS_{bwd}}$, the backwardly impacted set.

We follow Person et al. [30] to define three types of code changes: deleted, added, and modified. Our computation of the two impacted sets consists of several steps.

Algorithm 3 Forward and Backward Change-impact Analysis.

```
    Δ_diff ← Diff (P, P');
    Δ_map ← Map (P, P', Δ_diff);
1:  ComputeForwardImpactedSet(P, P')
2:      AI_fwd ← { }; MI_fwd ← { }; DI_fwd ← { };
3:      for each ( inst ∈ Δ_diff )
4:          if ( inst is added )
5:              AI_fwd ← AI_fwd ∪ FwdDependencyAnalysis(P', inst);
6:          else if ( inst is modified )
7:              MI_fwd ← MI_fwd ∪ FwdDependencyAnalysis(P', inst);
8:          else if ( inst is deleted )
9:              impacted ← FwdDependencyAnalysis(P, inst);
10:             for each ( st ∈ impacted )
11:                 st' ← QueryMap (Δ_map, st);
12:                 DI_fwd ← DI_fwd ∪ FwdDependencyAnalysis(P', st');
13:     return AI_fwd ∪ MI_fwd ∪ DI_fwd;

14: ComputeBackwardImpactedSet(P, P')
15:     AI_bwd ← { }; MI_bwd ← { }; DI_bwd ← { };
16:     for each ( inst ∈ Δ_diff )
17:         if ( inst is added )
18:             AI_bwd ← AI_bwd ∪ BwdDependencyAnalysis(P', inst);
19:         else if ( inst is modified )
20:             MI_bwd ← MI_bwd ∪ BwdDependencyAnalysis(P', inst);
21:         else if ( inst is deleted )
22:             impacted ← BwdDependencyAnalysis(P, inst);
23:             for each ( st ∈ impacted )
24:                 st' ← QueryMap (Δ_map, st);
25:                 DI_bwd ← DI_bwd ∪ BwdDependencyAnalysis(P', st');
26:     return DI_bwd ∪ MI_bwd ∪ DI_bwd;
```

First, we compare P and P' using a lightweight *diff* tool that computes the set Δ_{diff} of changed instructions (added, deleted, or modified). Since the remaining instructions exist in both programs, we construct a map Δ_{map} that maps every unchanged instruction $inst \in P$ to its counterpart $inst' \in P'$.

Second, for each added instruction, denoted $inst_{add} \in \Delta_{diff}$, we perform a forward control- and data-dependency analysis in P' to identify all instructions depending on $inst_{add}$ (Line 5). Details of this analysis are presented in the next subsection. We also perform a backward control- and data-dependency analysis in P' to identify all instructions that $inst_{add}$ depends on (Line 18). We denote the set of instructions as AI, represented separately as $\mathsf{AI_{fwd}}$ and $\mathsf{AI_{bwd}}$.

Third, for each modified instruction, denoted $inst_{mod} \in \Delta_{diff}$, we perform a forward control- and data-dependency analysis in P' to identify the instructions depending on $inst_{mod}$ (Line 7). We also perform a backward control- and data-dependency analysis to identify all instructions that $inst_{mod}$ depends on (Line 20). We denote the set of instructions as MI.

Fourth, for each deleted instruction $inst_{del} \in \Delta_{diff}$, we perform the forward control- and data-dependency analysis to compute the set of instructions depending on $inst_{del}$ (Line 9). We also perform the backward control- and data-dependency analysis to compute the set of instructions that $inst_{del}$ depends on (Line 22). For each instruction in this set, which is in program P, we retrieve its counterpart in P' by querying the Δ_{map}; the results form a new set DI.

Finally, the union of AI, MI, and DI forms the complete set of impacted instructions, denoted $\mathsf{IS_{fwd}}$ and $\mathsf{IS_{bwd}}$, respectively.

```
/**  [Thread 1]  **/
1:  x += 2;
2:  z = x + 1;
3:  y = x - 1;
4:  if (z>0)
5:      z = 0;
6:  else
7:      z-;
```

```
/**  [Thread 2]  **/
8:  z++;
9:  x -= 2;
10: if (x==0)
11:     y += 1; //modified
12: else
13:     z++;
14: assert(y != 2);
```

Figure 6: Example for our new change-impact analysis.

```
int x = 0;
void thread1() {
  int t1 = x;
  create(thread2);
}
void thread2() {
  x = 5;
}
```

Figure 7: Example for false data-dependencies across threads.

Algorithm 3 shows the actual pseudocode formalizing the above descriptions. For ease of comprehension, we have divided the computation of $\mathsf{IS}_{\mathsf{fwd}}$ and $\mathsf{IS}_{\mathsf{bwd}}$ into two separate routines. These routines, in turn, rely on two subroutines (described in Section 5.2) to perform the inter-thread and inter-procedural control- and data-dependency analysis.

Example. Figure 6 shows a program P that, starting with $x = y = z = 0$, may violate the assertion on Line 14 by executing Lines 1-3 and then 9-11. To fix the violation, we plan to change Line 11 from `y+=1` to `y+=2` to obtain the new program P'. During the change-impact analysis, Δ_{diff} = {11}, and Δ_{map} = {1-1, 2-2, ..., 14-14}. Since the type of change is *modified*, we only need to compute MI. Specifically, from the forward analysis, we obtain $\mathsf{MI}_{\mathsf{fwd}}$ = {11, 14}, which means the modification may affect Lines 11 and 14. From the backward analysis, we obtain $\mathsf{MI}_{\mathsf{bwd}}$ = {1, 3, 9, 10, 11}, which means they may affect the statement on Line 11. This is because Line 11 is control-dependent on Line 10 due to variable x, and data-dependent on Lines 3 and 11 due to variable y. Line 10, in turn, is data-dependent on Lines 1 and 9.

5.2 Computing the Dependency Relations

The dependency relations are computed by an inter-thread and inter-procedural static analysis. We follow [8, 15] to compute the control-dependencies using post-dominance, and data-dependencies by the transitive closure of use-def chains. Our main contributions, however, are reasoning about these dependencies in the concurrent setting (which also works on sequential programs), and adapting them to the forward/backward change-impact analysis.

We say that a statement s_2 is control-dependent on s_1 if the computation of s_1 determines whether s_2 is executed. For example, in `if(c)x++;` the statement `x++` is control-dependent on `if(c)` (specifically, on the value of the predicate `c`). On the other hand, s_4 is data-dependent on s_3 if the computation of s_3 influences the computation of s_4. For example, in `a=x;b=a+y;` the statement `b=a+y` is data-dependent on the statement `a=x` since the value of `a` determines the value of `b`.

To be conservative, our baseline dependency analysis is flow-insensitive, which has the advantage of being scalable and considering all ordering of statements. Since any statement from any thread can effectively execute at any time, this over-approximates the actual scheduling constraints, thereby ensuring the soundness of our analysis for multithreaded programs. However, using a flow-insensitive analysis, while sound, may result in false dependencies across threads. Consider the program in Figure 7: thread one reads the value of `x` and then creates thread two which writes to `x`. In a flow-insensitive analysis, the read in thread one is data-dependent on the write in thread two. But, the write can never be visible to thread one, since thread two does not exist until after the read.

To capture this situation, we augment our baseline dependency analysis with a *happen-before* relation. We say that a statement s_1 happens before a statement s_2 if on all program executions s_1 executes before s_2, e.g., the statement `create(thread2)` happens before `x = 5`. Toward this end, we refine the data-dependency analysis as follows: if s_1 happens-before s_2 then s_1 must not be data-dependent on s_2. This approach is comparable to recent works on using happens-before to refine data race detection [28, 27]. It is sound because the happens-before relation ensures there does not exist a program path from s_2 to s_1, and thus s_1 cannot witness the effect of s_2. Currently, we deduce happens-before constraints statically from the thread creation sites.

In the implementation, we adopt the Datalog-based declarative program analysis framework [44, 22, 2]: We first build CTRLDEP and DATADEP relations, where $(a, b) \in$ DATADEP means the variable a is data-dependent on b. We traverse the control flow graph to generate the set of input items for these relations. We use the structure of each individual instruction to determine the control- and data-dependency relations associated with it. For brevity, we show only how we handle the binary operation $r = op\, v_1\, v_2$, where r, v_1, and v_2 are variables and op is an operator. In this case, the input items to DATADEP are (r, v_1) and (r, v_2).

Similarly, we provide input items to the happens-before (HB) relation from thread creation sites. Within a thread, we determine the HB relation using dominance and reachability on the control-flow graph. Specifically, if s_1 dominates s_2 and s_1 is not reachable from s_2, then s_1 happens-before s_2. Dominance ensures that all paths to s_2 contain s_1; reachability ensures that there is no path from s_2 to s_1. All in all, they ensure s_1 always occurs before s_2.

We compute the transitive closure of CTRLDEP and DATADEP relations while using the happens-before relation to filter the false dependencies. Finally, the forward (resp. backward) dependency analysis on some statement s is the forward closure from s of the combination of the control- and data-dependency relations.

6. SUMMARY-BASED REDUNDANT PATH PRUNING

The second important component of our incremental analysis is pruning of redundant executions. In this section, we explain how to compute execution summaries in P and use them in the new program P'. First, during the symbolic execution of P, we summarize all the explored executions in a table, denoted PS, where each entry $\mathsf{PS}[s]$ stores a logical formula that represents all explored executions (suffixes) starting from s. Then, during the symbolic execution of P', we leverage our backward change-impact analysis to decide if these summaries can be carried over to P'.

6.1 Computing Execution Summaries

We construct the summary $\mathsf{PS}[s]$, at each state s, based on the weakest precondition (WP) computation [6]. The WP is defined with respect to a predicate ϕ and an execution π. It can be regarded as a form of Craig's interpolant [26, 16, 4], to explain why the execution cannot reach bad states. When an explored execution ends at an *assert(c)* statement, we compute the WP of c along this execution; otherwise, we compute the WP of `true`.

DEFINITION 1. *The weakest precondition of the predicate ϕ with respect to a sequence of instructions is defined as follows:*

- *For an assignment t: v :=exp, $WP(t, \phi) = \phi[exp/v]$, which is the substitution of v by exp in ϕ;*

```
1:  int AltPress:=0; Meter:=2
    procedure UPDATE(int PedalPos, int
    BSwitch, int PedalCmd)
2:    if PedalPos<=0 then //(modified)
3:      PedalCmd += 1
4:    else if PedelPos == 1 then
5:      PedalCmd += 2
6:    else PedalCmd = PedalPos
7:
8:    PedalCmd = PedalCmd + 1
9:
10:    if BSwitch == 0 then
11:      Meter = 1
12:    else if BSwitch == 1 then
13:      Meter = 2
14:
15:    if PedalCmd == 2 then
16:      AltPress = 0
17:    else if PedalCmd == 3 then
18:      AltPress = 1/4
19:    else AltPress == 1/2
20:
```

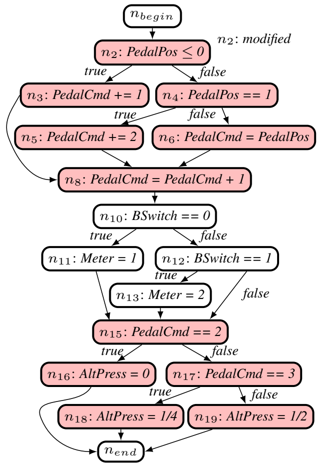

Figure 8: The WBS example taken from DiSE [30].

- *For a branching statement* t : `assume(c)` , $WP(t, \phi) = \phi \wedge c$*; and*
- *For a sequence of instructions, denoted* t_1 **;** t_2*,* $WP(t_1; t_2, \phi) = WP(t_1, WP(t_2, \phi))$*.*

Following Guo et al. [12], we compute the execution summary by merging the WPs at the pivot points as follows.

- The weakest precondition at a branching pivot point (b-PP) s, with outgoing edges to $s^1, \ldots, s^k$ and conditions $c_1, \ldots, c_k$, is defined as follows:

$$wp[s] := \bigvee_{1 \leq i \leq k} (c_i \wedge wp[s^i]) ,$$

  where each $wp[s^i]$ is the weakest precondition at state s^i.
- The weakest precondition at an interleaving pivot point (i-PP) s, with outgoing edges to $s^1, \ldots, s^k$, is defined as follows:

$$wp[s] := \bigwedge_{1 \leq i \leq k} wp[s^i] .$$

  This is an underapproximation since the precise merging would require an enumeration of all possible interleavings, which is too costly for the summary-based pruning. Nevertheless, in practice, this underapproximated summary often suffices for eliminating redundant executions.

Finally, the execution summary $\mathsf{PS}[s]$ at node s is computed as the union of the weakest preconditions along all explored executions starting from s.

Consider the WBS example in Figure 8, whose control flow graph is shown on the right-hand side. The baseline symbolic execution procedure needs to explore all 21 paths. Following the method described above, the execution summaries computed for the program P can be found in Table 1. For example, the summary for node n_{17}, denoted $\mathsf{PS}[n_{17}]$, is the union of ($PedalCmd$==3)∧$\mathsf{PS}[n_{18}]$) and ($PedalCmd$ ≠3)∧$\mathsf{PS}[n_{19}]$).

Prior to using the summary table computed in P in the new program P', we need to check if recent code changes have invalidated some of these summaries. If the answer is no, we can safely reuse them to prune away redundant executions in P'. For example, in Figure 5, since we changed only Line 1, i.e., from *if (x>0)* to *if (x>=0)*, the weakest precondition computation is not affected at all other nodes except for n_1. In other words, we can reuse the previously computed summaries at these nodes.

Table 1: Execution summaries computed for P in *Conc-iSE*.

Entry	Summary
$\mathsf{PS}[n_{19}]$	true
$\mathsf{PS}[n_{18}]$	true
$\mathsf{PS}[n_{17}]$	$((PedalCmd{==}3) \wedge \mathsf{PS}[n_{18}]) \vee ((PedalCmd \neq 3) \wedge \mathsf{PS}[n_{19}])$ = true
$\mathsf{PS}[n_{16}]$	true
$\mathsf{PS}[n_{15}]$	$((PedalCmd{==}2) \wedge \mathsf{PS}[n_{17}]) \vee ((PedalCmd \neq 2) \wedge \mathsf{PS}[n_{17}]))$ = true
$\mathsf{PS}[n_{13}]$	= $\mathsf{PS}[n_{15}][2/Meter]$ = true
$\mathsf{PS}[n_{12}]$	$((BSwitch{==}1) \wedge \mathsf{PS}[n_{15}]) \vee ((BSwitch \neq 1) \wedge \mathsf{PS}[n_{15}])$ = true
$\mathsf{PS}[n_{11}]$	= $\mathsf{PS}[n_{15}][1/Meter]$ = true
$\mathsf{PS}[n_{10}]$	$((BSwitch{==}0) \wedge \mathsf{PS}[n_{11}]) \vee ((BSwitch \neq 0) \wedge \mathsf{PS}[n_{12}])$ = true
$\mathsf{PS}[n_8]$	= $\mathsf{PS}[n_{10}][(PedalCmd + 1)/PedalCmd]$ = true
$\mathsf{PS}[n_6]$	= $\mathsf{PS}[n_8][PedalPos/PedalCmd]$ = true
$\mathsf{PS}[n_5]$	= $\mathsf{PS}[n_8][(PedalCmd + 2)/PedalCmd]$ = true
$\mathsf{PS}[n_4]$	$((PedalPos{==}1) \wedge \mathsf{PS}[n_5]) \vee ((PedalPos \neq 1) \wedge \mathsf{PS}[n_6])$ = true
$\mathsf{PS}[n_3]$	= $\mathsf{PS}[n_8][(PedalCmd + 1)/PedalCmd]$ = true
$\mathsf{PS}[n_2]$	$((PedalPos \leq 0) \wedge \mathsf{PS}[n_3]) \vee ((PedalPos{>}0) \wedge \mathsf{PS}[n_4])$ = true

Table 2: Comparing the paths explored by DiSE and *Conc-iSE*.

π	Explored by DiSE	Explored by *Conc-iSE*
1	$\{n_2, n_3, n_8, n_{10}, n_{11}, n_{15}, n_{16}\}$	partial (up to n_3)
2	$\{n_2, n_3, n_8, n_{10}, n_{11}, n_{15}, n_{17}, n_{18}\}$	skipped
3	$\{n_2, n_3, n_8, n_{10}, n_{11}, n_{15}, n_{17}, n_{19}\}$	skipped
4	$\{n_2, n_4, n_5, n_8, n_{10}, n_{11}, n_{15}, n_{16}\}$	partial (up to n_4)
5	$\{n_2, n_4, n_5, n_8, n_{10}, n_{11}, n_{15}, n_{17}, n_{18}\}$	skipped
6	$\{n_2, n_4, n_5, n_8, n_{10}, n_{11}, n_{15}, n_{17}, n_{19}\}$	skipped
7	$\{n_2, n_4, n_6, n_8, n_{10}, n_{11}, n_{15}, n_{16}\}$	skipped

6.2 Pruning with Execution Summaries

Our method for leveraging the summaries to prune away redundant executions has been shown on Lines 27–29 in Algorithm 2. Here, $pcon$ represents the set of forwardly reachable states, while $\neg\mathsf{PS}[s]$ represents the set of states that may lead to some previously unexplored errors. If the intersection is empty, however, there is no need to continue the current execution beyond s. In the actual implementation, the validity of ($pcon \implies \mathsf{PS}[s]$) is decided by checking the satisfiability of its negation, ($pcon \wedge \neg\mathsf{PS}[s]$), which can be solved efficiently by an SMT solver.

To demonstrate the advantages of our method, we show how it works on the WBS example from DiSE [30]. Since DiSE works only for sequential programs, the WBS example in Figure 8 is a sequential program and our method assumes it has a single thread. In WBS, the only code change is on Line 2, from (*PedalPos* == 0) to (*PedalPos* <= 0). The red rounded rectangles represent the impacted CFG nodes in P', while the white rounded rectangles represent nodes that are not impacted by the change. The baseline symbolic execution procedure needs to explore all 21 paths whereas DiSE only needs to explore 7 paths (Table 2), due to the reduction based on its forward impact analysis. That is, the nodes n_{10}, n_{11}, n_{12} and n_{13} are not affected by the code change at n_2.

However, there is still redundancy among the 7 paths explored by DiSE. As shown in the third column of Table 2, certain common subpaths are explored repeatedly. For example, $\{n_8, n_{10}, n_{11}, n_{15}, n_{16}\}$ is an already-explored subpath in π_1 but it is re-explored in π_4 and π_7, also, $\{n_{10}, n_{11}, n_{15}, n_{17}, n_{18}\}$ is an already-explored subpath in π_2 but it is re-explored in π_5, and $\{n_{10}, n_{11}, n_{15}, n_{17}, n_{19}\}$ is an already-explored subpath in π_3 but it is re-explored in π_6. In contrast, our new method can reduce the seven executions further down to 2 executions (Column 3 in Table 2).

Specifically, during the symbolic execution of P, we incrementally compute the summaries at n_{17}, n_{15}, n_{10}, n_4, n_2, and Table 1 shows the summary table of P in terms of these locations.

Then, in the symbolic execution of the new program P', we first apply the forward change-impact analysis for the modification in Line 2, and then apply our backward change-impact analysis, which indicates that the summary is invalid only at node n_2 (immediately before Line 2); for all other nodes, we can safely reuse the summaries since these nodes are not in the backward slice of n_2.

By checking the validity of $pcon[s] \implies \mathsf{PS}[s]$ for all nodes except for n_2 during the execution, we can reduce the seven runs further down to two partial runs. More specifically, the execution on P' starts by visiting n_2. As the summary at n_2 is invalid (since it is in the backward impacted-set), the execution continues exploring without checking the summary. Consider that the *true* branch of n_2 is first selected; execution proceeds until reaching the next assignment statement at n_3. Noticing that the summary at n_3 is still valid and $(pcon[n_3] \wedge \neg\mathsf{PS}[n_3])$ = $(PedalPos{\leq}0) \wedge \neg\mathsf{true}$ = false, the execution stops here, generates the first partial run $\{n_2, n_3\}$, and backtracks to n_2.

Next, the *false* branch of n_2 is selected and the execution runs until the following branch statement at n_4. As $(pcon[n_4] \wedge \neg\mathsf{PS}[n_4])$ = $(PedalPos{>}0) \wedge \neg\mathsf{true}$ = false, the execution also stops, generates the second partial run $\{n_2, n_4\}$, then backtracks to n_2. Since both outgoing edges of n_2 are explored and n_2 is the entry of the program, the whole execution on P' terminates.

Therefore, the two runs in P' explored by our method are $\pi_1 = \{n_2, n_3\}$ and $\pi_4 = \{n_2, n_4\}$, shown in Column 3 of Table 2.

7. EXPERIMENTS

We have implemented the proposed method in a software tool named *Conc-iSE*, which builds upon LLVM [23] and *Cloud9* [5]. *Cloud9* relies on the KLEE symbolic virtual machine [3] as backend. We extended *Cloud9* to robustly handle POSIX threads; the original implementation only coarsely considered different thread interleavings at blocking operations. In contrast, our symbolic execution procedure schedules threads at a finer granularity (e.g., the shared memory accesses) and ensures that all interleavings are systematically explored. Furthermore, we implemented the dynamic partial-order reduction (DPOR) algorithm [9], which *Cloud9* does not originally support. In addition, we have implemented our forward and backward change impact analysis to provide guidance to our incremental symbolic execution algorithm. We also implemented a flow-insensitive pointer analysis for multi-threaded programs. Our dependency analysis is constraint-based and directly works on the LLVM bit-code. We use the Z3's μZ [14] fix-point solver to compute the fix-point of the Datalog constraints.

To share the summary information between program versions, we deployed the Memcached Distributed Cache as an external persistent storage for the execution summaries. The summaries are computed and encoded in KLEE KQuery formula format during the symbolic execution. After the execution of the original program P, they are serialized as binary character sequences for Memcached storage. Before running the new program P', they are loaded into main memory and mapped to the corresponding global control locations. Based on the results of our backward change-impact analysis, we implemented a *summary renewal* mechanism to check if the summary of a location has been invalidated by recent code changes, and reset it to false in that situation.

7.1 Subjects and Methodology

We have conducted experiments on two sets of benchmarks. The first set consists of multi-threaded C programs randomly chosen from the Software Verification Competition benchmark [37] and benchmarks from [7]. The second set consists of three real-world applications, each with five different versions: they are lock-free data structure implementations (*nbds-list*, *nbds-skiplist* and *nbds-hashtable*) from [29]. Each of these benchmark programs has between 50 to 2,500 lines of code, with a total of 14 applications, 70 different versions, and 34,926 lines of code. Each benchmark program is first compiled to LLVM bit-code by Clang, before given to the symbolic execution engine.

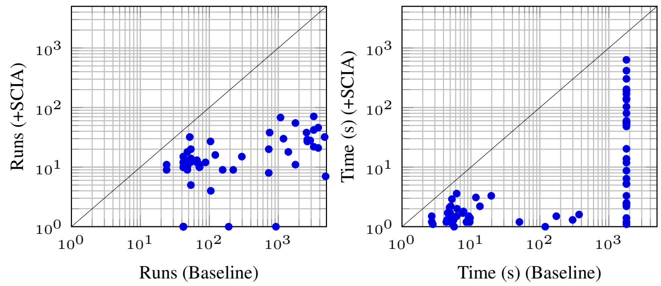

Figure 9: *Conc-iSE* (+SCIA) versus the Baseline algorithm.

For C programs from [37, 7, 20], since there are no different versions available online, we manually made three types of mutants to the programs, acting as modified, deleted and added statements. For the real-world applications from [29], we studied the evolution history from the code repository, and used real updates committed by their developers as the changes to those programs.

7.2 Experimental Results

Table 3 summarizes the experimental results of our evaluation. The program name, version, lines of code, number of changes, percentage of code impacted, and the number of threads for each program are shown in Columns 1–6. Columns 7–14 compare the experimental performance of four different methods in terms of the number of explored executions (runs) and the time in seconds.

Baseline denotes the baseline symbolic execution procedure in Algorithm 1, *+DPOR* denotes baseline symbolic execution augmented with dynamic partial order reduction, *+CIA* denotes a variant of *Conc-iSE*, which augments baseline symbolic execution with DPOR and pruning based on the forward change, but without the backward summary-based pruning. Finally, *+SCIA* denotes the full-blown implementation of *Conc-iSE*, which augments *+CIA* with the backward summary-based pruning. In all methods, the static analysis time and summary computation time (if applicable) are included in the total execution time. We used a maximum time of 30 minutes (1,800 seconds) for all experiments.

In the remainder of this section, we present the results in more detail to answer the following research questions:

1. How effective is our *Incremental Symbolic Execution* algorithm?
2. How does it compare to state-of-the-art POR techniques?
3. How effective is the backward summary-based pruning?

First, we compare the performance of *Baseline* and *+SCIA* with the two scatter plots in Figure 9. The x-axis denotes the execution time (or number of runs) of the baseline symbolic execution, while the y-axis denotes the execution time (or number of runs) of our new method (*+SCIA*). In the scatter plots, each dot represents a benchmark program, and the dots below the diagonal lines are the winning cases of our new method. From Figure 9, we see that our new method can significantly reduce the number of runs explored by symbolic execution as well as the total execution time. In many cases, our new method can finish the execution in seconds while the baseline algorithm does not stop after 30 minutes.

Second, we compare the performance of *+DPOR* and *+SCIA* with the two scatter plots in Figure 10. Our goal is to show how much performance improvement was achieved by our new method over *+DPOR* alone. Similarly, dots below the diagonal lines are the winning cases of our new method (*+SCIA*). Again, our new method brings significant performance improvement compared to *+DPOR*. However, there are some test cases where *+SCIA* spent slightly longer time, despite that it has the same or a smaller number of runs. This is due to the overhead of static analysis, summary

Table 3: Comparing the two variants of *Conc-iSE* (+CIA and +SCIA) with baseline symbolic execution.

						Existing Methods				*Conc-iSE* (new)			
						Baseline		+DPOR		+CIA		+SCIA	
Name	Version	LOC	# Changes	Impacted (%)	Threads	# Runs	Time (s)	# Runs	Time (s)	# Runs	Time (s)	# Runs	Time (s)
	v1	65	1	0.0		924	16.6	48	0.8	1	0.3	1	0.3
	v2	66	2	10.6		–	>1800	142	2.0	142	2.2	22	0.9
fibbench [37]	v3	67	2	13.6	2	–	>1800	628	12.1	628	12.0	34	2.4
	v4	67	2	17.9		–	>1800	3943	160.3	3943	161.4	39	2.5
	v5	68	3	2.9		–	>1800	1420	30.3	10	0.4	7	0.3
	v1	68	1	8.8		749	14.2	106	1.8	106	1.7	38	0.9
	v2	69	1	10.1		5838	370.1	208	3.5	208	3.4	81	1.6
account [37]	v3	70	3	14.3	3	1773	50.5	168	2.8	168	2.7	55	1.2
	v4	70	1	6.6		1773	47.8	168	2.7	14	0.5	11	0.4
	v5	71	2	6.6		13407	1642.4	325	5.3	11	0.4	9	0.3
	v1	58	1	10.3		156	2.4	12	0.4	12	0.4	9	0.3
	v2	59	2	11.9		1399	36.1	43	0.8	43	0.8	18	0.5
lazy01 [37]	v3	61	4	11.5	3	8313	624.1	71	1.2	71	1.2	18	0.5
	v4	62	2	1.6		8313	625.3	71	1.1	2	0.3	2	0.1
	v5	61	4	13.1		–	>1800	211	3.1	179	2.5	26	0.6
	v1	85	1	22.4		–	>1800	729	29.6	729	30.5	33	20.3
	v2	85	1	16.5		–	>1800	81	2.5	5	0.4	5	0.4
indexer [37]	v3	86	2	23.3	2	–	>1800	90	2.5	90	2.6	30	5.2
	v4	87	2	2.3		–	>1800	90	2.5	1	0.3	1	0.3
	v5	88	2	22.7		–	>1800	1314	41.2	1314	43.7	563	53.9
	v1	59	1	5.1		191	2.7	36	0.7	1	0.2	1	0.3
	v2	60	3	13.3		105	1.6	10	0.4	4	0.3	4	0.3
readreadwrite [37]	v3	63	3	12.7	3	728	13.7	34	0.7	34	0.8	20	0.5
	v4	63	1	12.7		728	14.0	34	0.7	9	0.3	8	0.3
	v5	67	5	19.1		5444	175.1	101	1.6	22	0.6	18	0.5
	v1	65	2	9.2		88	1.4	37	0.7	37	0.7	12	0.4
	v2	67	1	9.0		296	4.3	117	1.7	46	0.8	15	0.4
stateful01 [37]	v3	68	2	10.3	2	3267	120.8	675	11.4	327	4.8	22	0.5
	v4	68	1	16.2		3267	119.6	675	10.1	675	9.9	71	1.0
	v5	68	1	16.2		3267	121.3	675	8.9	675	8.9	42	0.7
	v1	94	1	6.4		1190	17.8	38	0.7	34	0.6	30	0.5
	v2	92	2	6.5		222	2.7	15	0.5	11	0.4	9	1.2
reorder [37]	v3	94	2	7.4	2	2903	72.1	61	1.2	38	0.7	28	0.5
	v4	96	2	7.4		4698	176.1	125	1.9	125	1.9	32	0.7
	v5	97	3	7.2		9557	273.1	68	1.2	53	0.9	39	0.8
	v1	141	1	2.8		4862	286.8	101	1.6	7	0.3	7	0.3
	v2	142	1	5.6		5878	298.4	148	2.2	148	2.4	79	1.3
twostage3 [37]	v3	141	2	5.7	3	2636	97.8	101	1.6	60	1.1	27	0.6
	v4	141	1	7.1		2636	96.0	69	1.4	37	0.8	29	0.6
	v5	141	1	5.1		2568	94.7	188	3.2	123	2.3	38	0.7
	v1	73	1	20.5		–	>1800	12473	171.3	2	0.3	2	0.3
	v2	74	2	27.4		–	>1800	13434	197.6	150	2.2	67	1.4
szymanski [7]	v3	73	1	20.5	2	–	>1800	10180	136.7	73	1.3	61	1.1
	v4	73	1	26.7		–	>1800	14365	207.7	591	8.9	79	2.2
	v5	73	1	20.0		–	>1800	–	>1800	73	1.3	61	1.2
	v1	128	2	25.8		–	>1800	2112	31.7	1739	25.1	287	8.7
	v2	130	2	22.1		–	>1800	2292	34.6	1133	16.4	223	6.4
bluetooth [7]	v3	130	1	22.3	2	–	>1800	2324	35.3	1154	16.5	276	5.3
	v4	131	5	38.2		–	>1800	2617	40.6	2617	41.5	532	13.8
	v5	133	3	39.1		–	>1800	2437	38.5	2437	36.1	417	11.9
	v1	115	1	26.9		52	0.9	52	0.9	52	0.9	32	0.8
	v2	116	2	15.5		1077	19.7	277	4.1	277	4.1	68	3.3
circularbuf [7]	v3	116	1	6.9	2	3794	171.8	770	14.3	126	1.9	21	0.5
	v4	118	2	15.3		3794	173.1	2916	105.6	462	7.4	46	1.5
	v5	117	1	28.2		–	>1800	924	17.8	924	18.1	102	3.3
	v1	1168	5	9.2		–	>1800	1724	433.9	501	223.3	422	136.1
	v2	1624	3	1.9		–	>1800	898	117.3	10	141.6	10	141.6
nbds-list [29]	v3	1626	4	5.2	2	–	>1800	4660	701.6	503	102.9	503	103.2
	v4	1887	5	3.5		–	>1800	6007	698.9	35	90.7	14	80.4
	v5	1885	3	5.0		–	>1800	1304	160.7	198	73.2	175	53.1
	v1	1734	2	10.3		–	>1800	–	>1800	1874	263.6	1266	202.7
	v2	2095	2	3.0		–	>1800	4645	228.0	284	61.6	180	56.5
nbds-skiplist [29]	v3	2095	2	3.2	2	–	>1800	–	>1800	299	61.9	223	59.9
	v4	2100	3	0.4		–	>1800	7508	266.3	5	48.3	5	48.2
	v5	2101	1	2.5		–	>1800	–	>1800	550	65.6	417	56.3
	v1	2234	1	0.3		–	>1800	4818	218.6	9	170.1	9	169.5
	v2	2322	2	8.6		–	>1800	–	>1800	2686	650.8	2686	632.6
nbds-hashtable [29]	v3	2375	2	7.3	2	–	>1800	–	>1800	1684	440.5	1453	416.1
	v4	2418	2	2.7		–	>1800	9474	730.8	612	258.8	431	190.3
	v5	2422	2	4.6		–	>1800	17556	1396.2	849	337.1	763	303.5
Total		**34,926**					**70,585**		**17,149**		**3,478**		**2,816**

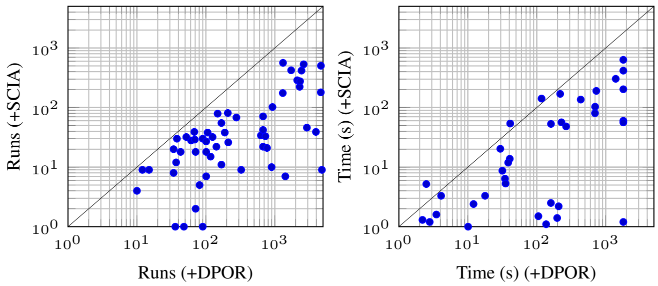

Figure 10: *Conc-iSE* (+SCIA) versus Baseline (+DPOR).

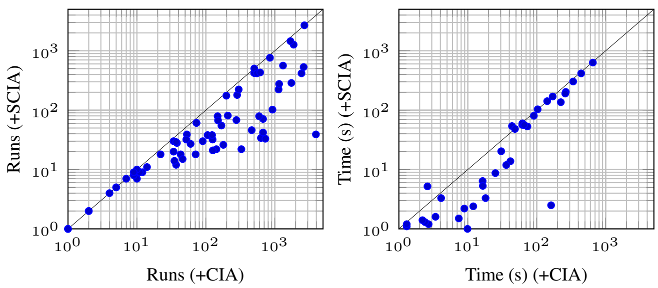

Figure 11: *Conc-iSE* variants: (+CIA) versus (+SCIA).

computation, as well as the pruning, which makes the total execution slower than *+DPOR*. But, overall, the run time of *+SCIA* versus *+DPOR* is 83% smaller.

Finally, we compare the two *Conc-iSE* variants (*+CIA* and *+SCIA*) in Figure 11. These scatter plots show the effect of execution summaries during an incremental analysis. Similar to the previous cases, sometimes the summary-based pruning technique is not able to provide a significant reduction, thereby causing the runtime to be slightly higher; this usually occurs when the backward impact analysis causes many summaries to be removed. Nonetheless, for most test cases, it is able to have a significant reduction in the number of runs, which in turn leads to a significant reduction in time.

*Discussion.*Fundamentally, an incremental analysis is only applicable when the code modification affects a subset of the entire program: if the entire program is modified then the incremental analysis degenerates to the non-incremental one. Therefore, our technique is suitable in a software development environment where the correctness of frequent but small code changes is checked before they are committed to the central repository. In our experiments, the code modifications from the *nbds* application are all developer-made modifications. Furthermore, in these real-world applications, code modifications typically affected around 0.3% to 10.3% of the entire program. Such code changes are small enough to allow *Conc-iSE* to be effective, although it remains an open question whether they reflect the majority of the software development scenarios in practice. Another interesting problem is when to schedule tests, e.g., as in Herzig et al. [13], which is an orthogonal but closely related problem.

8. RELATED WORK

Change-impact analysis [45] has been widely applied in software testing and verification. The existing incremental symbolic execution tool, DiSE [30], uses an intra-procedural static change-impact analysis and then leverages it to reduce the cost of symbolic execution. The extension of DiSE, named iDiSE [32], improves it in two ways: by making the change-impact analysis inter-procedural, and by using dynamic calling-context information to increase accuracy. Yang et al. [46] extend DiSE to a property-guided symbolic-execution procedure for checking assertions in evolving programs.

Change-impact analysis has been used in the context of program verification as well. For example, Backes et al. [1] use a change-impact analysis to improve the functional equivalence checking in regression verification. Specifically, the change-impact is used to focus on the equivalence checking of affected portions of the code. Similarly, SymDiff [21] focuses on proving assertions in the context of regression verification.

However, none of these previous techniques were designed for concurrent programs: they all target sequential software. Their extension to concurrent programs remains non-trivial due to the inherent difficulties in analyzing thread interferences. Our new technique, in contrast, is the first incremental symbolic execution for concurrent programs.

SimRT [49] is a regression-testing tool for multithreaded programs targeting data-races. It compares the two program versions syntactically to identify a set of affected variables, and then construct a list of potential data races to test. During the testing phase, SimRT prioritizes the selection of existing test cases and visiting the most program points of the affected variables to speed up data-race detection. CAPP [17] uses a change-impact analysis to prioritize scheduler preemptions at impacted code points to detect concurrency bugs. However, CAPP only manipulates the prioritized thread scheduling rules with fixed data inputs.

Furthermore, SimRT and CAPP focus on test selection and prioritization as opposed to generating new tests. In contrast, our method uses symbolic execution to generate new tests.

RECONTEST [38] is a regression testing technique to select new thread interleavings that are more likely to trigger concurrency bugs caused by recent code changes. Specifically, it computes the affected code statements by comparing dynamic execution traces on the two program versions. Then, at each program point of the impacted set, it identifies problematic memory access patterns [39, 36] and use them to compute alternative interleavings, e.g., by reordering these concurrent memory accesses. While RECONTEST has the capability of exploring new thread schedules, it relies on user-provided data inputs. In contrast, we use symbolic execution to generate new data inputs as well as new thread schedules.

We build upon prior works on constructing weakest-precondition and similar interpolation-based execution summaries during symbolic execution [26, 16, 4, 48, 12]. There is also a large body of work on symbolic analysis of concurrent software using SMT solvers [42, 40, 41, 19, 34, 18, 35, 10, 25]. However, these prior works target a single program version. In contrast, we leverage the summary computed in the previous program version to prune redundant executions in the new program version.

9. CONCLUSION

We have presented *Conc-iSE*, an incremental symbolic execution algorithm for concurrent programs. Our new change-impact analysis is both inter-thread and inter-procedural, capable of more accurately identifying instructions affected from code changes between two closely related program versions. We also showed how summaries computed from the previous program can be used to prune away redundant runs during symbolic execution of the new program. We implemented our method and evaluated it on a large set of multithreaded programs. Our experiments show that the new method can significantly reduce the runtime cost when compared with the state-of-the-art symbolic execution techniques.

10. ACKNOWLEDGMENTS

This work was primarily supported by the NSF under grants CCF-1149454, CCF-1405697, and CCF-1500024. Partial support was provided by the ONR under grant N00014-13-1-0527.

11. REFERENCES

[1] J. D. Backes, S. Person, N. Rungta, and O. Tkachuk. Regression verification using impact summaries. In *International SPIN Workshop on Model Checking Software*, pages 99–116, 2013.

[2] M. Bravenboer and Y. Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 243–262, 2009.

[3] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 209–224, 2008.

[4] D. Chu and J. Jaffar. A framework to synergize partial order reduction with state interpolation. In *International Haifa Verification Conference*, pages 171–187, 2014.

[5] L. Ciortea, C. Zamfir, S. Bucur, V. Chipounov, and G. Candea. Cloud9: a software testing service. *Operating Systems Review*, 43(4):5–10, 2009.

[6] E. Dijkstra. *A Discipline of Programming*. Prentice Hall, NJ, 1976.

[7] A. Farzan, A. Holzer, N. Razavi, and H. Veith. Con2colic testing. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 37–47, 2013.

[8] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, July 1987.

[9] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 110–121, 2005.

[10] M. K. Ganai, N. Arora, C. Wang, A. Gupta, and G. Balakrishnan. BEST: A symbolic testing tool for predicting multi-threaded program failures. In *IEEE/ACM International Conference On Automated Software Engineering*, pages 596–599, 2011.

[11] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*. Springer, 1996.

[12] S. Guo, M. Kusano, C. Wang, Z. Yang, and A. Gupta. Assertion guided symbolic execution of multithreaded programs. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, 2015.

[13] K. Herzig, M. Greiler, J. Czerwonka, and B. Murphy. The art of testing less without sacrificing quality. In *International Conference on Software Engineering*, pages 483–493, Piscataway, NJ, USA, 2015. IEEE Press.

[14] K. Hoder, N. Bjørner, and L. de Moura. muZ - an efficient engine for fixed points with constraints. In *International Conference on Computer Aided Verification*, volume 6806 of *Lecture Notes in Computer Science*, pages 457–462, 2011.

[15] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Program. Lang. Syst.*, 12(1):26–60, Jan. 1990.

[16] J. Jaffar, V. Murali, and J. A. Navas. Boosting concolic testing via interpolation. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 48–58, 2013.

[17] V. Jagannath, Q. Luo, and D. Marinov. Change-aware preemption prioritization. In *International Symposium on Software Testing and Analysis*, 2011.

[18] V. Kahlon and C. Wang. Universal Causality Graphs: A precise happens-before model for detecting bugs in concurrent programs. In *International Conference on Computer Aided Verification*, pages 434–449, 2010.

[19] S. Kundu, M. K. Ganai, and C. Wang. CONTESSA: Concurrency testing augmented with symbolic analysis. In *International Conference on Computer Aided Verification*, pages 127–131, 2010.

[20] M. Kusano and C. Wang. Assertion guided abstraction: a cooperative optimization for dynamic partial order reduction. In *IEEE/ACM International Conference On Automated Software Engineering*, pages 175–186, 2014.

[21] S. K. Lahiri, K. L. McMillan, R. Sharma, and C. Hawblitzel. Differential assertion checking. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 345–355, 2013.

[22] M. S. Lam, J. Whaley, V. B. Livshits, M. C. Martin, D. Avots, M. Carbin, and C. Unkel. Context-sensitive program analysis as database queries. In *ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 1–12, 2005.

[23] C. Lattner and V. Adve. The LLVM Compiler Framework and Infrastructure Tutorial. In *LCPC'04 Mini Workshop on Compiler Research Infrastructures*, West Lafayette, Indiana, Sep 2004.

[24] S. Lehnert. A taxonomy for software change impact analysis. In *International Workshop on Principles of Software Evolution*, pages 41–50, 2011.

[25] L. Li and C. Wang. Dynamic analysis and debugging of binary code for security applications. In *International Conference on Runtime Verification*, pages 403–423, 2013.

[26] K. L. McMillan. Lazy annotation for program testing and verification. In *International Conference on Computer Aided Verification*, pages 104–118, 2010.

[27] M. Naik and A. Aiken. Conditional must not aliasing for static race detection. *SIGPLAN Not.*, 42(1):327–338, Jan. 2007.

[28] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for java. *SIGPLAN Not.*, 41(6):308–319, June 2006.

[29] Non-blocking data structures. URL: https://code.google.com/p/nbds/.

[30] S. Person, G. Yang, N. Rungta, and S. Khurshid. Directed incremental symbolic execution. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 504–515, 2011.

[31] S. Raghavan, R. Rohana, D. Leon, A. Podgurski, and V. Augustine. Dex: A semantic-graph differencing tool for studying changes in large code bases. In *IEEE International Conference on Software Maintenance*, pages 188–197, 2004.

[32] N. Rungta, S. Person, and J. Branchaud. A change impact analysis to characterize evolving program behaviors. In *IEEE International Conference on Software Maintenance*, pages 109–118, 2012.

[33] K. Sen. *Scalable Automated Methods for Dynamic Program Analysis*. PhD thesis, UIUC, 2006.

[34] N. Sinha and C. Wang. Staged concurrent program analysis. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 47–56, 2010.

[35] N. Sinha and C. Wang. On interference abstractions. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 423–434, 2011.

[36] W. N. Sumner, C. Hammer, and J. Dolby. Marathon: Detecting atomic-set serializability violations with conflict graphs. In *International Conference on Runtime Verification*, pages 161–176, 2012.

[37] SV-COMP. 2015 software verification competition. URL: http://sv-comp.sosy-lab.org/2015/, 2015.

[38] V. Terragni, S.-C. Cheung, and C. Zhang. RECONTEST: Effective regression testing of concurrent programs. In *International Conference on Software Engineering*, 2015.

[39] M. Vaziri, F. Tip, and J. Dolby. Associating synchronization constraints with data in an object-oriented language. In J. G. Morrisett and S. L. P. Jones, editors, *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 334–345. ACM, 2006.

[40] C. Wang, S. Chaudhuri, A. Gupta, and Y. Yang. Symbolic pruning of concurrent program executions. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 23–32, 2009.

[41] C. Wang, S. Kundu, M. Ganai, and A. Gupta. Symbolic predictive analysis for concurrent programs. In *International Symposium on Formal Methods*, pages 256–272, 2009.

[42] C. Wang, Z. Yang, V. Kahlon, and A. Gupta. Peephole partial order reduction. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 382–396, 2008.

[43] M. Weiser. Program slicing. In *International Conference on Software Engineering*, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[44] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 131–144, 2004.

[45] J. W. Wilkerson. A software change impact analysis taxonomy. In *IEEE International Conference on Software Maintenance*, pages 625–628, 2012.

[46] G. Yang, S. Khurshid, S. Person, and N. Rungta. Property differencing for incremental checking. In *International Conference on Software Engineering*, pages 1059–1070, 2014.

[47] W. Yang. Identifying syntactic differences between two programs. *Softw., Pract. Exper.*, 21(7):739–755, 1991.

[48] Q. Yi, Z. Yang, S. Guo, C. Wang, J. Liu, and C. Zhao. Postconditioned symbolic execution. In *IEEE International Conference on Software Testing, Verification and Validation*, pages 1–10, 2015.

[49] T. Yu, W. Srisa-an, and G. Rothermel. SimRT: an automated framework to support regression testing for data races. In *International Conference on Software Engineering*, pages 48–59, 2014.