Assertion Guided Symbolic Execution of Multithreaded Programs

Shengjian Guo
Department of ECE
Virginia Tech

Markus Kusano
Department of ECE
Virginia Tech

Chao Wang
Department of ECE
Virginia Tech

Zijiang Yang
Department of CS
Western Michigan University

Aarti Gupta
Department of CS
Princeton University

ABSTRACT

Symbolic execution has emerged as a powerful technique for systematic testing of sequential and multithreaded programs. However, its application is limited by the high computational cost of covering all feasible intra-thread paths and inter-thread interleavings. We propose a new assertion guided pruning framework that identifies executions guaranteed not to lead to an error state and removes them during symbolic execution. By summarizing the reasons why previously explored executions cannot reach an error state and using the information to prune redundant executions in the future, we can soundly reduce the search space exponentially. We also use static concurrent program slicing and heuristic minimization of symbolic constraints to further reduce the computational overhead. We have implemented our method in the *Cloud9* symbolic execution tool and evaluated it on a large set of multithreaded C/C++ programs. Our experiments show that the new method can reduce the overall computational cost significantly.

Categories and Subject Descriptors

F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs; D.2.4 [**Software Engineering**]: Software/Program Verification

Keywords

Symbolic execution, test generation, concurrency, partial order reduction, weakest precondition

1. INTRODUCTION

The past decade has seen exciting developments on symbolic execution of both sequential [19, 44, 49, 8] and concurrent programs [42, 39, 14, 5]. However, existing methods are still limited in their capability of mitigating the *state space explosion*. That is, the number of paths in each thread may be exponential to the number of branch conditions, and the number of thread interleavings may be exponential to the number of concurrent operations. Many techniques have been proposed to address this problem, including the use of function summaries [18], interpolation [34, 23, 62], static analysis [7], and coverage metrics [14]. In this paper, we propose a new and complementary method, which is designed specifically for pruning redundant executions in multithreaded programs where the properties under verification are expressed as assertions.

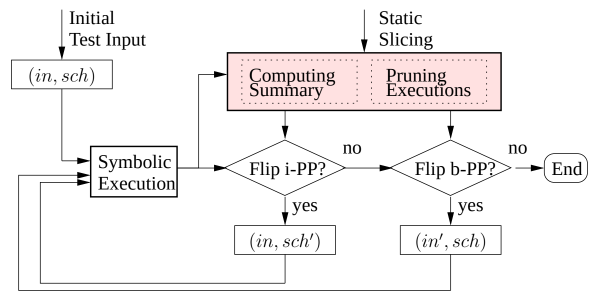


Figure 1. Our assertion guided redundancy pruning framework.

Our *assertion guided* symbolic execution framework focuses on identifying and eliminating executions that are guaranteed to be redundant for checking assertions. Assertions can be used to model a wide variety of interesting properties, ranging from logic and numerical errors, to memory safety and concurrency errors, and has been the focus of many software verification projects. When semantic errors of the program are modeled as simple code reachability, i.e., the reachability of a bad state guarded by the assertion condition, we can concentrate on exploring potentially failure-inducing executions as opposed to all feasible executions of the program. This is particularly attractive in the presence of *concurrency*, since it becomes possible to uniformly handle the exploration of both intra-thread execution paths and inter-thread interleavings leading to a simple but more powerful analysis algorithm.

The overall flow of our new method is illustrated in Figure 1: the shaded block represents our addition and the remainder illustrates the classic symbolic execution procedure for multithreaded programs [42]. Specifically, given a program P and some symbolic input variables, the procedure explores the feasible executions of the program systematically, e.g., in a depth-first search order.

Starting with an initial test (in, sch) consisting of program inputs and thread schedule, the method first produces a concrete execution followed by a symbolic execution. Then, it tries to generate a new test by flipping a prior decision at either an thread interleaving pivot point (i-PP) or a local branch pivot point (b-PP). The new test is denoted by either (in, sch') or (in', sch), depending on whether changes are made to the thread schedule (sch') or data input (in'), respectively. The iterative procedure terminates when no new test can be generated. State explosion occurs because it has to explore the combined space of data inputs and thread schedules where each individual execution may be unique, i.e., it leads to a different program state.

We extend the baseline symbolic execution procedure by adding a new constraint-based pruning block shown in Figure 1. Our method centers around the idea of summarizing the reasons why the bad state is unreachable via previously explored executions, and leveraging such information to avoid similarly futile executions. Specifically, at each global control location n, we use a predicate summary (PS) constraint to capture the *weakest preconditions* [13] of the assertion condition along all explored executions starting from n. Therefore, $\mathsf{PS}[n]$ captures the reason why prior executions are not able to violate the assertion. Whenever symbolic state n is reached again through another execution path, we check if the new *path condition* is subsumed by $\mathsf{PS}[n]$. If so, we can safely backtrack from n since extending the execution beyond n would never lead to a bad state.

Our method for pruning redundant executions can be viewed as a way of systematically exploring an abstract search space defined by a set of predicates [4] which, in this case, are extracted from the assertion. Although the concrete search space may be arbitrarily large, the abstract search space can be significantly smaller. In this sense, our method is similar to *predicate abstraction* [21] in model checking except that the latter requires constructing *a priori* a finite-state model from the actual software code whereas our method directly works on the software code while leveraging the predicates to eliminate redundant executions.

Our method is complementary to standard partial order reduction (POR) techniques in that it relies on property-specific information to reduce the state space. But, POR techniques typically do not target particular states. We will show through experiments that our new method can indeed eliminate a different class of redundant executions from those eliminated by state-of-the-art POR techniques, such as dynamic partial order reduction (DPOR) [16]. Toward this end, since DPOR is an elegant but delicate algorithm that can easily be made unsound without taking great care in the implementation [60], a main technical challenge in our work is to make sure our new pruning method does not interfere with DPOR or make it less effective.

Our method differs from the prior works by Wachter et al. [52], and Chu and Jaffar [9], which extended the well-known framework of lazy abstraction with interpolants by McMillan [34] to multithreaded programs. One main difference is that our computation of predicate summaries is significantly more general than existing methods, especially at the thread interleaving pivot points, where we merge summaries from multiple execution paths to form a combined summary. Another main difference is in the integration of property specific pruning with partial order reduction. Both existing methods implemented a variant of the symbolic partial order reduction algorithm by Kahlon et al. [26] whereas we integrate our predicate summary-based pruning method with the more scalable DPOR algorithm.

We have implemented our method in *Cloud9* [11], a state-of-the-art symbolic execution tool built upon LLVM and KLEE [8], to handle multithreaded C/C++ programs. We have implemented an inter-procedural static program slicing algorithm [22], executed prior to symbolic execution, to further reduce the search space. We have also implemented heuristic based minimizations of predicate summary constraints during symbolic execution to reduce the computational overhead. In both cases, the main technical challenge is to ensure the overall algorithm remains sound in the presence of such optimizations. We have conducted experiments on a set of standard multithreaded C/C++ applications. Our results show that the new method can reduce the number of explored executions as well as the overall run time significantly.

To sum up, this paper makes the following contributions:

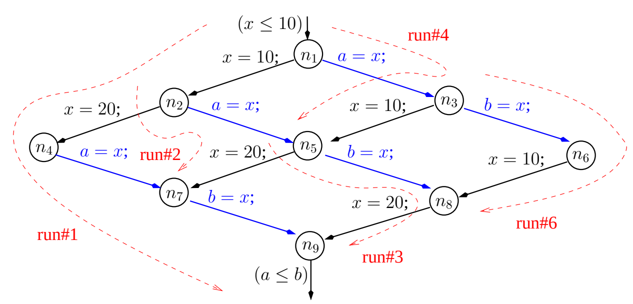
- We propose an assertion guided symbolic execution method to identify and eliminate redundant executions in multithreaded programs to reduce the overall computational cost.
- We implement our method in a state-of-the-art symbolic execution tool while ensuring it does not interfere with the popular DPOR algorithm or make it less effective.
- We demonstrate through experiments that our new method can indeed achieve a significant performance improvement on public benchmarks.

The remainder of this paper is organized as follows. First, we illustrate our new method through examples in Section 2, then establish the notation and review the baseline symbolic execution algorithm in Section 3. We present our method for summarizing explored executions in Section 4 and pruning redundant executions in Section 5. We present optimization techniques in Section 6 and experimental results in Section 7. We review related work in Section 8 and finally give our conclusions in Section 9.

2. MOTIVATING EXAMPLES

In this section, we illustrate the high-level ideas in our method using examples. Consider the example in Figure 2, which has two threads T_1 and T_2, a global variable x, and two local variables a and b. The initial value of x is a symbolic input which can be any integer value. We want to check if the assertion fails and, if so, compute a failure-inducing test input.

```
                x = symbolic(V);
                if(x>10) return;
----[T1]----------------------------[T2]----
   x = 10;                          a = x;
   x = 20;                          b = x;
--------------------------------------------
                assert( a<=b )
```



Run 1: `if(x<=10)x=10;x=20;a=x;b=x;` leads to `(a=20,b=20)`.
Run 2: `if(x<=10)x=10;a=x;x=20;b=x;` leads to `(a=10,b=20)`.
Run 3: `if(x<=10)x=10;a=x;b=x;x=20;` leads to `(a=10,b=10)`.
Run 4: `if(x<=10)a=x;x=10;x=20;b=x;` leads to `(a=V ,b=20)`.
Run 5: `if(x<=10)a=x;x=10;b=x;x=20;` leads to `(a=V ,b=10)`.
Run 6: `if(x<=10)a=x;b=x;x=10;x=20;` leads to `(a=V ,b=V )`.

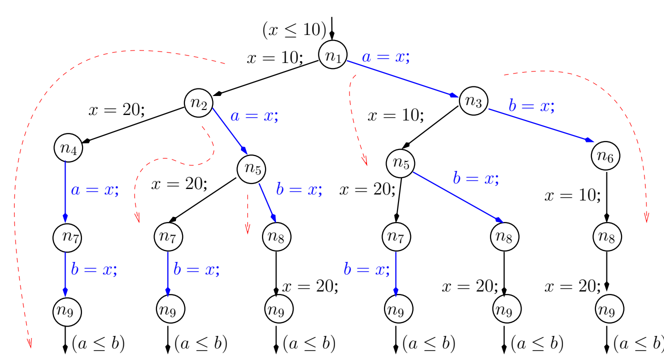


Figure 2. Our new method only needs to explore one full run and four partial runs, as opposed to all six runs by existing methods.

The program has six distinct executions, each leading to a different final state defined by the values of a and b. According to the theory of partial order reduction [16], they belong to six different equivalence classes [32], as each has a different final state. However, exploring all six executions is not necessary for the purpose of checking the assertion, since some of these executions share the same reason why they cannot reach the bad state. Our new method can reduce the exploration from six executions to one full execution together with four partial executions, as illustrated by the red dotted lines in Figure 2.

Our method first extracts a set of predicates by computing the weakest preconditions of the assertion condition along the explored executions. These predicates are then combined at the merge points (in the graph) to form a succinct summary that captures the reason why the bad state has not been reached via executions starting from these merge points. During subsequent symbolic execution iterations, our method needs to explore only those executions that have not be covered by these predicates, thereby leading to a sound reduction of the search space.

Now, we provide a step-by-step explanation of how our method works on this example:

- Run 1 is the first and only execution fully explored by our new method, which goes through nodes n_1, n_2, n_4, n_7 in the graph in Figure 2 before executing `b=x;if(a<=b)`. Since it does not violation the assertion, we summarize the reason at n_9 and n_7, respectively, as follows: $\mathsf{PS}[n_9] = (a \leq b)$ and $\mathsf{PS}[n_7] = (a \leq x)$. That is, as long as $(a \leq x)$ holds at node n_7, it would be impossible for the execution to reach the bad state.

- Run 2 goes through nodes n_1, n_2, n_5 before reaching n_7, where its new path condition is $pcon[n_7] = (V \leq 10)$ and symbolic memory is $\mathcal{M} =$`(a=10,x=20)`. Since $pcon[n_7] \rightarrow \mathsf{PS}[n_7]$ under $\mathcal{M}$, meaning the set of reachable states falls inside $\mathsf{PS}[n_7]$, continuing the current execution from n_7 would never lead to a bad state. Therefore, we skip the remainder of this execution.

- Run 3 goes through nodes n_1, n_2, n_5, n_8 before reaching n_9, where its path condition again falls within $\mathsf{PS}[n_9]$. We skip the remainder of this execution and update the summary at n_8 and n_5 as follows: $\mathsf{PS}[n_8] = (a \leq b)$ and $\mathsf{PS}[n_5] = wp[n_7] \wedge wp[n_8] = (a \leq 20) \wedge (a \leq x)$. By conjoining the weakest preconditions along both interleavings $n_5 \rightarrow n_7$ and $n_5 \rightarrow n_8$, we capture the summary common to both interleavings.

- Run 4 goes through nodes n_1, n_3 before reaching n_5, with the new path condition $pcon[n_5] = (V \leq 10)$ and symbolic memory $\mathcal{M} =$`(a=V,x=10)`. Since $pcon[n_5] \rightarrow \mathsf{PS}[n_5]$ under $\mathcal{M}$, we skip the remainder of this execution, which would have led to Run 4 and Run 5 if it is allowed to continue.

- Run 6 goes through nodes n_1, n_3, n_6 before reaching n_8, where the new path condition falls within $\mathsf{PS}[n_8]$. Therefore, we skip the remainder of this execution.

- At this moment, our method has completed the exploration.

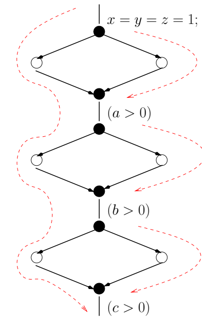
Note that we *conjoin* weakest preconditions from different interleavings at i-PP nodes such as n_5, but *union* weakest preconditions from different thread-local paths at b-PP nodes (see Section 4.) Also note that the amount of reduction achieved by our method depends on the program structure as well as the location of the assertion. For example, if we change `if(x>10)` to `if(x>11)`, our method would have to explore Run 5 instead of skipping it because $pcon[n_5] = (V \leq 11)$ would no longer be subsumed by $\mathsf{PS}[n_5] = (V \leq 10)$.

```
        x = y = z = 1;

---[T1]--------------[T2]---
  a = x;            x = 10;
----------------------------
        assert(a>0)

---[T1]-------------[T3]----
  b = y;            y = 10;
----------------------------
        assert(b>0)

---[T1]-------------[T4]----
  c = z;            z = 10;
----------------------------
        assert(c>0)
```



* Run 1: `a=x;x=10;if(a>0);b=y;y=10;if(b>0);c=z;z=10;if(c>0)`.
* Run 2: `a=x;x=10;if(a>0);b=y;y=10;if(b>0);z=10;c=z;if(c>0)`.
Run 3: `a=x;x=10;if(a>0);y=10;b=y;if(b>0);c=z;z=10;if(c>0)`.
* Run 4: `a=x;x=10;if(a>0);y=10;b=y;if(b>0);z=10;c=z;if(c>0)`.
* Run 5: `x=10;a=x;if(a>0);b=y;y=10;if(b>0);c=z;z=10;if(c>0)`.
Run 6: `x=10;a=x;if(a>0);b=y;y=10;if(b>0);z=10;c=z;if(c>0)`.
Run 7: `x=10;a=x;if(a>0);y=10;b=y;if(b>0);c=z;z=10;if(c>0)`.
Run 8: `x=10;a=x;if(a>0);y=10;b=y;if(b>0);z=10;c=z;if(c>0)`.

Figure 3. Our new method can reduce the number of executions from 2^k down to $(k+1)$.

The running example demonstrates that our method differs from standard partial order reduction techniques such as DPOR [16] which could not prune away any of the six interleavings. Furthermore, our method also differs from the stateful state space exploration techniques commonly used in model checking, which record the forward reachable states explicitly during exploration to prevent visiting them again. Such methods would not be effective for the example in Figure 2 either because each of the six executions leads to a distinct state. In contrast, our new method can achieve a significant reduction due to its use of property specific information as guidance. In this sense, our new method is a *property directed* reduction, whereas the aforementioned POR techniques are *property agnostic*.

However, it can be tricky to combine our pruning method with the state-of-the-art DPOR algorithm. The main advantage of DPOR over static POR techniques lies in its dynamic update of backtrack sets, which uses runtime information to compute the dependency relation between shared variable accesses. Without taking any additional measure, pruning redundant executions may interfere with the dynamic update of backtrack sets in DPOR. Consider run 4 in Figure 2 as an example. If the execution is allowed to complete, when `b=x` is executed, thread T_2 will be added to the backtrack set of node n_3. However, if run 4 is terminated pre-maturely at node n_5 due to our predicate summary-based pruning, thread T_2 would not be added to the backtrack set of node n_3 since `b=x` has been skipped. As a result, the DPOR algorithm would not explore run 6. Therefore, integrating DPOR with property specific pruning is a challenging task. We present our solution to this problem in Section 5.2.

Our computation of predicate summaries at the thread interleaving merge point n_5 in Figure 2 shows that it is different from the prior work by Wachter et al. [52], and Chu and Jaffar [9]. Specifically, we combine the summaries from all outgoing edges by conjoining them together, whereas existing methods do not merge interpolants at these i-PP nodes. Furthermore, our method differs from these existing methods in that they both implemented a symbolic POR whereas our method is integrated with the more scalable DPOR algorithm.

Now, we use the example in Figure 3 to demonstrate that our new method has the potential to achieve an exponential reduction. In this contrived example, the interleaving of instructions in `{a=x, x=10}` is completely independent from `{b=y, y=10}` and `{c=z, z=10}`. Exploring all feasible executions results in 2^3 runs, each of which leads to a different final state. However, based on the abstract search space induced by the assertions, our new method can reduce the exploration of eight runs down to one full run together with three partial runs, as marked by the '*' symbol in Figure 3. To further generalize the example, a program with k independent code segments would have 2^k distinct interleavings, which can be reduced by our method to $(k+1)$ executions.

3. PRELIMINARIES

We establish the notation and review the baseline symbolic execution algorithm for multithreaded programs in this section.

3.1 Multithreaded Programs

For ease of presentation, we consider a simple imperative language with integer variables, assignments, and if-else statements only. We elide the details for handling of complex language features such as pointers, recursion, and system calls in symbolic execution since these are orthogonal issues addressed previously by many symbolic execution tools [8, 11]. A multithreaded program P consists of a set of threads $\{T_1 \dots T_m\}$, where each thread, T_i, is a sequential program. Threads share a set of *global* variables. Each thread also has a set of *local* variables.

Let st be an instruction in a thread with the thread index tid. Let *event* $e = \langle tid, l, st, l' \rangle$ be an execution instance of st, where l and l' are locations in the thread before and after executing the instance of st. If the same instruction is executed more than once, e.g., when it is in a loop or a recursive function call, we make copies of l, st, l' to make them unique for each event. Conceptually, this corresponds to unrolling loops and recursive calls. A *global control state* of the multithreaded program is a tuple $s = \langle l_1, \dots, l_m \rangle$, where each l_i is a location in T_i. We regard a global control state as an *abstract* state implicitly containing all concrete states that have the same thread locations but potentially different values of the local and global variables.

Without loss of generality, we assume that every assertion of the form `assert(c)` is transformed to `if(!c)abort`. We use a special event **abort** to denote faulty program termination and **halt** to denote normal program termination. Let v_l denote a local variable, v_g denote a global variable, $cond_l$ denote a local condition, and exp_l denote an local expression. In addition to **abort** and **halt**, each instruction st in an event may have one of the following types:

- α-operation, which is a local assignment $v_l := exp_l$;
- β-operation, which is a local branch `assume`$(cond_l)$;
- γ-operation, which is a global operation defined as follows:
  - γ-I is a global write $v_g := exp_l$ or read $v_l := v_g$;
  - γ-II is a thread synchronization operations.

For each `if(c)-else` statement, we use `assume`(c) to denote the execution of then-branch, and `assume(`$\neg c$`)` to denote the execution of else-branch. Without loss of generality, we assume that all if-else conditions use only local variables or local copies of global variables [17]. For thread synchronizations, we focus on mutex locks and condition variables since they are frequently used in mainstream multithreaded programming environments such as C, C++, and Java. Specifically, we consider the following types of γ-II operations: thread creation, thread join, lock, unlock, signal, and wait. If other thread synchronizations or blocking operations are used they can be modeled similarly as γ-II events.

During the execution of the program, γ-operations are the thread interleaving points whereas β-operations are thread-local branching points. Both contribute to the path and interleaving explosion. In contrast, α-operations are local and thus invisible to other threads; they do not contribute directly to the path and interleaving explosion.

A *concrete* execution of the multithreaded program is characterized by $\pi = (in, sch)$, where in is the data input and sch is the thread schedule corresponding to the total order of events $e_1 \dots e_n$. The corresponding *symbolic* execution is denoted by $(*, sch)$, where the $*$ indicates the data input is kept symbolic and thus may take any value. Each execution of the program P can be represented by a finite word $\{\alpha, \beta, \gamma\}^* \{\textbf{halt}, \textbf{abort}\}$. If the execution ends with **halt** it is a *normal* execution. If the execution ends with **abort** it is a *faulty* execution.

```
a1: a=x++;
a2: if(a==0) A1;
    else Ā1;
a3: a=y++;
a4: if(a==0) A2;
    else Ā2;
a5:
---  [T1] ---
```

```
b1: b=x++;
b2: if(b==0) B1;
    else B̄1;
b3: b=y++;
b4: if(b==0) B2;
    else B̄2;
b5:
---  [T2] ---
```

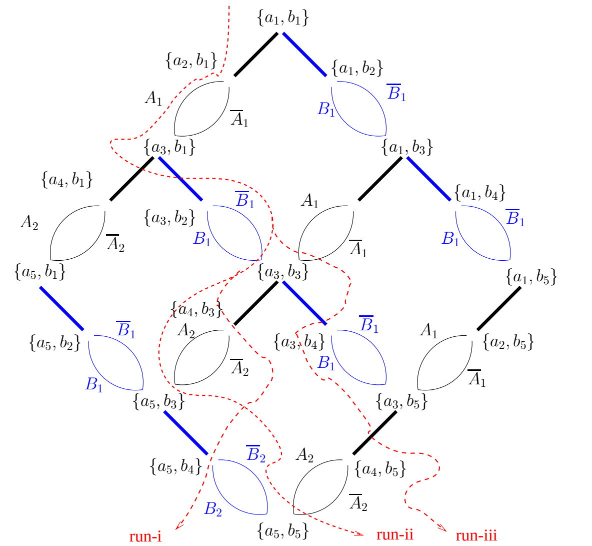


Figure 4. A two-threaded program and its generalized interleaving graph (GIG). Black edges represent events from thread T_1 and blue edges represent events from thread T_2.

3.2 Generalized Interleaving Graph (GIG)

The set of all possible executions of a multithreaded program can be captured by a *generalized interleaving graph (GIG)*, where nodes are global control states and edges are events. The root node corresponds to the (symbolic) initial state. The leaf nodes correspond to normal or faulty ends of the execution. Each internal node may have:

- one outgoing edge corresponding to an α-operation;
- two outgoing edges corresponding to a β-operation; or
- k outgoing edges where $k \geq 2$ is the number of enabled γ-operations from different threads.

We call a node with more than one outgoing edge a *pivot point*.

- If the pivot point corresponds to β-operations we call it a *branching pivot point (**b-PP**)*.
- If the pivot point corresponds to γ-operations we call it a thread *interleaving pivot point (**i-PP**)*.

Figure 4 shows an example program and its GIG. For simplicity, we assume that `a=x++` is atomic on the execution platform. The root node (a_1, b_1) corresponds to the starting points of the two threads. The terminal node (a_5, b_5) corresponds to the end of the two threads. Nodes such as (a_1, b_1) are i-PP nodes, where we can execute either thread 1 which leads to (a_2, b_1), or thread 2 which leads to (a_1, b_2). In contrast, nodes such as (a_2, b_1) are b-PP nodes, where we can take either the `assume(`$a = 0$`)` branch, leading to the code segment A_1, or the `assume(`$a \neq 0$`)` branch, leading to the code segment $\overline{A}_1$.

Note that the GIG does not have loop-back edges since the GIG paths represent unrolled executions. Furthermore, pointers, aliasing, and function calls have been resolved as well during execution. However, a GIG may have branches, which makes it significantly different from the typical thread interleaving graph used in the partial order reduction literature.

As is typical in symbolic execution algorithms, we focus on only a finite set of executions and assume that each execution has a finite length. Typically, the user of a symbolic execution tool needs to construct a proper testing environment that satisfies the above assumption. In KLEE [8] and *Cloud9* [11], for example, the user may achieve this by bounding the size of the symbolic input thereby restricting the execution to a fixed number of paths of finite lengths.

3.3 Symbolic Execution of Multithreaded Programs

We present the baseline symbolic execution procedure for multithreaded programs in Algorithm 1 following Sen et al. [42]. The recursive procedure EXPLORE is invoked with the symbolic initial state s_0. Inside the procedure, we differentiate among three scenarios based on whether s, the current state, is an *i-PP* node, a *b-PP* node, or a non-branching node.

If s is an *i-PP* node where multiple γ-operations are enabled, we recursively explore the next γ event from each thread. If s is a *b-PP* node where multiple sequential branches are feasible, we recursively explore each branch. If s is a non-branching node, we explore the unique next event. The current execution ends if s is a leaf node (normal_end_state, faulty_end_state) or an infeasible_state, at which point we return from EXPLORE(s) by popping the state s from the stack S.

Algorithm 1 Baseline Symbolic Execution.

Initially: Stack $S = \{s_0\}$; run EXPLORE(s_0) with the symbolic initial state s_0.
EXPLORE(s)
  S.push(s);
  if (s is an i-PP node)
    while ($\exists t \in (s.enabled \setminus s.done)$)
      $s' \leftarrow$ NEXTSTATE(s, t);
      EXPLORE(s');
      $s.done \leftarrow s.done \cup \{t\}$;
  else if (s is a b-PP node) {
    while ($\exists t \in (s.branch \setminus s.done)$)
      $s' \leftarrow$ NEXTSTATE(s, t);
      EXPLORE(s');
      $s.done \leftarrow s.done \cup \{t\}$;
  else if (s is an internal node)
    $t \leftarrow s.next$;
    $s' \leftarrow$ NEXTSTATE(s, t);
    EXPLORE(s');
  S.pop();
NEXTSTATE(s, t)
  let $s = \langle pcon, \mathcal{M}, enabled, branch, done\rangle$;
  if (t is **halt**)
    $s' \leftarrow$ normal_end_state;
  else if (t is **abort**)
    $s' \leftarrow$ faulty_end_state;
  else if (t is $\mathbf{assume(c)}$)
    if ($s.pcon$ is unsatisfiable under $\mathcal{M}$)
      $s' \leftarrow$ infeasible_state;
    else
      $s' \leftarrow \langle pcon \wedge c, \mathcal{M}\rangle$;
  else if (t is assignment $\mathbf{v := exp}$)
    $s' \leftarrow \langle pcon, \mathcal{M}[exp/v]\rangle$;
  return s';

Each state $s \in S$ is a tuple $\langle pcon, \mathcal{M}, enabled, branch, done\rangle$, where $pcon$ is the path condition for the execution to reach s from s_0, $\mathcal{M}$ is the symbolic memory map, $s.enabled$ is the set of γ-events when s is an i-PP node, $s.branch$ is the set of β-events when s is a b-PP node, and $s.done$ is the set of α or β events already explored from s by the recursive procedure. Initially, s_0 is set to $\langle true, \mathcal{M}_{init}\rangle$, where $true$ means the state is always reachable and $\mathcal{M}_{init}$ represents the initial content of the memory. The execution of each instruction t is carried out by NEXTSTATE(s, t) as follows:

- If t is **halt**, the execution ends normally.
- If t is **abort**, and $s.pcon$ is satisfiable under the current memory map $s.\mathcal{M}$, we have found an error.
- If t is **v:=exp**, we need to update the current memory map $\mathcal{M}$ by changing the content of v to exp.
- If t is **assume(c)**, we change the path condition to $(pcon \wedge c)$.

At each pivot point (i-PP or b-PP), we try to flip a decision made previously to compute a new execution. Let (in, sch) denote the current execution. By flipping the decision made previously at an i-PP node, we compute a new execution (in, sch'), where sch' is a permutation of the original thread schedule. In contrast, by flipping the decision made previously at a b-PP node, we compute a new execution (in', sch), where in' is a new data input. Note that in both cases, the newly computed execution will be the same as the original execution up to the flipped pivot point. After the flipping, the rest of the execution will be a free run.

As an example, consider the GIG in Figure 4, where the current execution is represented by the dotted line ***run-i***. Flipping at the b-PP node (a_4, b_3) would lead to the new execution labeled ***run-ii***, whereas flipping at the i-PP node (a_3, b_3) would lead to the new execution ***run-iii***.

4. SUMMARIZING THE EXPLORED EXECUTIONS

We first present our method for symbolically summarizing the reason why explored executions cannot reach the bad state. In the next section, we will leverage the information to prune away redundant executions.

Our method for summarizing the explored executions is based on the weakest precondition computation [13]. We differentiate the following two scenarios, depending on whether the execution encounters the assert statement or not.

- For each execution that encounters $assert(c)$ (and satisfies the condition c), we compute the weakest precondition of the predicate c along this execution.
- For each execution that does not encounter the assert statement at all, we compute the weakest precondition of the predicate `true` along this execution.

Since the weakest precondition is a form of Craig's interpolant [34], it provides a succinct explanation as to why the explored execution cannot reach the bad state guarded by $\neg c$.

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

- *For t:* `v:=exp`*, $WP(t, \phi) = \phi[exp/v]$;*
- *For t:* `assume(c)` *, $WP(t, \phi) = \phi \wedge c$; and*
- *For sequence t_1;t_2, $WP(t_1; t_2, \phi) = WP(t_1, WP(t_2, \phi))$.*

In the above definition, $\phi[exp/v]$ denotes the substitution of variable v in ϕ with exp. As an example, consider the execution path in the following table, which consists of three branch conditions and three assignments. Column 1 shows the control locations along the current path. Column 2 shows the sequence of instructions executed. Column 3 shows the weakest preconditions computed backwardly starting at l_6. Column 4 shows the rules applied during the computation.

Loc.	*Instruction*	*WP Computed*	*Rule Applied*
l_0	$if(a \le 0)$	$(a \le 0) \wedge (b \le 0) \wedge (c \le 0)$	$wp \wedge c$
l_1	$res := res + 1$	$(b \le 0) \wedge (c \le 0)$	$wp[exp/v]$
l_2	$if(b \le 0)$	$(b \le 0) \wedge (c \le 0)$	$wp \wedge c$
l_3	$res := res + 2$	$(c \le 0)$	$wp[exp/v]$
l_4	$if(c \le 0)$	$(c \le 0)$	$wp \wedge c$
l_5	$res := res + 3$	true	$wp[exp/v]$
l_6		true	*terminal*

4.1 Computing Predicate Summary at b-PP Nodes

Assume that the baseline symbolic execution procedure traverses the GIG in a depth-first search (DFS) order, meaning that it backtracks s, a branching pivot point (b-PP), only after exploring both outgoing edges $s \xrightarrow{assume(c)} s'$ and $s \xrightarrow{assume(\neg c)} s''$. This also includes the entire execution trees starting from these two edges. Let $wp[s']$ and $wp[s'']$ be the weakest preconditions computed from the two outgoing executions, respectively.

Following the classic definition of weakest precondition provided by Dijkstra [13], we compute the weakest precondition at the b-PP node s as follows:

$$wp[s] := (c \wedge wp[s']) \vee (\neg c \wedge wp[s'']) \ .$$

Then, we use $wp[s]$ computed from these outgoing edges to update the global predicate summary.

The predicate summary, $\mathsf{PS}[s]$, defined for each global control state s, is the union of all weakest preconditions along the outgoing edges. Recall that each node s may be visited by EXPLORE multiple times, presumably from different execution paths (from s_0 to s). Therefore, we maintain a global map PS and update each predicate summary entry $\mathsf{PS}[s]$ incrementally. Initially $\mathsf{PS}[s] = \mathsf{false}$ for every GIG node s. Then, we merge the newly computed $wp[s]$ to $\mathsf{PS}[s]$ every time EXPLORE backtracks from s.

The detailed method for updating the predicate summary is highlighted in blue in Algorithm 2, which follows the overall flow of Algorithm 1, except for the following two additions:

- We compute $wp[s]$ before the procedure backtracks from state s. At this moment, $wp[s]$ captures the set of all explored executions from s as a continuation of the current execution.
- We update the summary as follows: $\mathsf{PS}[s] = \mathsf{PS}[s] \vee wp[s]$. Here, $\mathsf{PS}[s]$ captures the set of execution trees as a continuation of all explored executions from s_0 to s, including $wp[s]$, which represents the newly explored execution tree.

Algorithm 2 Assertion Guided Symbolic Execution.

Initially: summary $\mathsf{PS}[n] = \mathsf{false}$ for all node n; stack $S = \{s_0\}$; run EXPLORE(s_0) with initial state s_0.

EXPLORE(s)
  S.push(s);
  if (s is an i-PP node)
    $wp[s] := \mathsf{true}$;
    while ($\exists t \in (s.enabled \setminus s.done)$)
      $s' \leftarrow$ NEXTSTATE(s, t);
      EXPLORE(s');
      $wp[s] \leftarrow wp[s] \wedge$ COMPUTEWP (s, t, s');
      $s.done \leftarrow s.done \cup \{t\}$;
  else if (s is a b-PP node)
    $wp[s] := \mathsf{false}$;
    while ($\exists t \in (s.branch \setminus s.done)$)
      $s' \leftarrow$ NEXTSTATE(s, t);
      EXPLORE(s');
      $wp[s] \leftarrow wp[s] \vee$ COMPUTEWP (s, t, s');
      $s.done \leftarrow s.done \cup \{t\}$;
  else if (s is an internal node)
    $t \leftarrow s.next$;
    $s' \leftarrow$ NEXTSTATE(s, t);
    EXPLORE(s');
    $wp[s] \leftarrow$ COMPUTEWP (s, t, s');
  else // end state
    $wp[s] \leftarrow \mathsf{true}$;
  $\mathsf{PS}[s] := \mathsf{PS}[s] \vee wp[s]$;
  S.pop();
COMPUTEWP(s, t, s')
  if (t is $assume(c)$)
    return ($wp[s'] \wedge c$);
  else if (t is assignment $v := exp$)
    return substitute($wp[s'], v, exp$);
  else
    return $wp[s']$;
NEXTSTATE(s, t)
  let s be tuple $\langle pcon, \mathcal{M}, enabled, branch, done \rangle$;
  if (t is **halt**)
    $s' \leftarrow$ normal_end_state;
  else if (t is **abort**)
    $s' \leftarrow$ faulty_end_state;
  else if (t is $\mathbf{assume(c)}$)
    if ($s.pcon$ is unsatisfiable under $\mathcal{M}$)
      $s' \leftarrow$ infeasible_state;
    else if ($pcon \rightarrow \mathsf{PS}[s]$)
      $s' \leftarrow$ early_termination_state;
    else
      $s' \leftarrow \langle pcon \wedge c, \mathcal{M} \rangle$;
  else if (t is assignment $\mathbf{v} := \mathbf{exp}$)
    $s' \leftarrow \langle pcon, \mathcal{M}[exp/v] \rangle$;
  return s';

4.2 Computing Predicate Summary at i-PP Nodes

In contrast to the straightforward computation of weakest precondition at the sequential merge point, the situation at the interleaving merge point is trickier. In fact, to the best of our knowledge, there does not exist a definition of weakest precondition in the literature for thread interleaving points.

A naive extension of Dijkstra's original definition would be inefficient since it leads to the explicit enumeration of all possible interleavings. For example, assume that an i-PP node has two outgoing edges $s \xrightarrow{\gamma_1} s'$ and $s \xrightarrow{\gamma_2} s''$, one may attempt to define the weakest precondition at node s as follows:

$$\big((\gamma_1 <_{hb} \gamma_2) \wedge wp[s']\big) \vee \big((\gamma_2 <_{hb} \gamma_1) \wedge wp[s'']\big) \ ,$$

where $(\gamma_1 <_{hb} \gamma_2)$ means that we choose to execute γ_1 before γ_2, $(\gamma_2 <_{hb} \gamma_1)$ means that we choose to execute γ_2 before γ_1, and $wp[s']$ and $wp[s'']$ are the weakest preconditions along the two interleavings, respectively.

Although the above definition serves the purpose of summarizing the weakest preconditions along all explored executions from s, it has a drawback: the size of $wp[s]$ computed in this way can quickly explode when there are a large number of threads. Recall that in a multithreaded program the number of outgoing edges at an i-PP node equals the number of enabled threads and the number of interleavings of k concurrent threads can be $k!$ in the worst case.

However, for the purpose of pruning redundant executions, the weakest precondition computation does not have to be precise to be effective. To mitigate the aforementioned interleaving explosion problem, we resort to the following definition, which can be viewed as an under-approximation of the naive definition:

$$wp[s] := \bigwedge_{1 \le i \le k} wp[s^k] \ ,$$

where each $wp[s^i]$ is the weakest precondition computed along one of the k outgoing edges of the form $s \xrightarrow{\gamma_i} s^i$, such that $1 \le i \le k$. Consider Figure 2 as an example. We compute the weakest precondition at node n_5 by conjoining weakest preconditions at the two successor nodes n_7 and n_8. That is, $wp[n5] = wp[n_7] \wedge wp[n_8] = (a \le 20) \wedge (a \le x)$.

For the purpose of pruning redundant executions, conjoining weakest preconditions from different interleavings at i-PP nodes is a sound approximation. Although it may not capture all the explored executions and thus fail to prune certain redundant executions, all the pruned executions are guaranteed to be redundant.

5. PRUNING THE REDUNDANT EXECUTIONS

We present our method for leveraging the predicate summaries to prune away redundant executions in this section.

5.1 Assertion Guided Pruning

To decide if we can skip executions starting from a global control state s where s has been visited by EXPLORE previously through some executions from s_0 to s, but is reached again through a new execution, we check whether the current path condition $s.pcon$ is subsumed by $\mathsf{PS}[s]$ under the current memory map $s.\mathcal{M}$. Intuitively, the path condition $s.pcon$ represents the set of states reachable along the current execution from s_0 to s, whereas $\mathsf{PS}[s]$ represents the set of states from which it is impossible to reach the bad state.

Within the NEXTSTATE procedure in Algorithm 2, we check for the pruning condition as follow:

- If $s.pcon \rightarrow \mathsf{PS}[s]$ holds under $s.\mathcal{M}$, extending the current execution beyond s would not lead to a bad state. Therefore, we backtrack immediately by setting s' as an *early termination state*.
- Otherwise, there *may* exist an extension of the current execution beyond s to reach the bad state. In this case, we need to continue the forward symbolic execution as usual.

The validity of $s.pcon \rightarrow \mathsf{PS}[s]$ can be decided by checking the satisfiability of $(s.pcon \wedge \neg\mathsf{PS}[s])$ using an SMT solver. That is, $s.pcon \rightarrow \mathsf{PS}[s]$ holds if and only if $(s.pcon \wedge \neg\mathsf{PS}[s])$ is unsatisfiable.

Our new pruning method is complementary to partial order reduction techniques. POR is a generic reduction that relies solely on commutativity between concurrent operations. Therefore, two executions are considered equivalent as long as they result in the same program state. Our new method, in contrast, uses assertions to guide the pruning. Therefore, even executions that result in different program states may still be regarded as equivalent.

Consider the GIG in Figure 4, which has 54 feasible executions. To make the presentation simple, we have assumed that `x++` is atomic in this example. However, note that `a1:a= x++` and `b1:b= x++` do not commute, because from a state where x=0, for instance, executing `a1;b1` leads to a=0,b=1,x=2, but executing `b1;a1` leads to a=1,b=0,x=2. As shown in Table 1, without applying any reduction technique, the program has a total of 54 distinct runs. Partial order reduction (POR) alone can reduce the 54 runs down to 34 runs. Our new predicate summary-based pruning method alone can reduce the 54 runs down to the 18 runs. Finally, applying both our method and POR can reduce the 54 runs down to 13 runs.

Table 1. Applying various reduction techniques to Figure 4.

Reduction Technique	Number of Paths
None	54
Partial order reduction (POR)	34
Our predicate summary-based pruning method	18
Both POR and our new pruning method	13

5.2 Interaction with DPOR

However, there is a caveat in combining our predicate summary-based pruning method with dynamic partial order reduction [16], because DPOR is a delicate algorithm that relies on the dynamic computation of the *backtrack sets*. Without taking precautions, naively pruning away redundant executions, even if they do not lead to the bad state, may deprive DPOR the opportunity to properly update its backtrack sets, thereby leading to unsound reduction.

As we have shown in Section 2, when the current execution is run 4 in Figure 2, by the time node n_5 is reached DPOR has not had the opportunity to update its backtrack set at n_3. Ideally, thread T_2 should be put into the backtrack set of n_3, that is, after EXPLORE backtracks to n_3, it should proceed to explore run 6.

However, since $n_5.pcon \rightarrow \mathsf{PS}[n_5]$ along run 4, our pruning method would force EXPLORE to backtrack from n_5, thereby skipping the remainder of run 4 and run 5. Here, the technical challenge is how to properly update the backtrack set at node n_3 before EXPLORE backtracks from n_5.

Fortunately, similar problems were encountered during the development of stateful DPOR algorithms [60]. In this work, we follow the solution by Yang et al. [60]. We maintain two global tables, $RVar[s]$ and $WVar[s]$, for each global control state s. The $RVar$ table stores the set of global variables that have been read by some thread during previously explored executions starting from s. Similarly, the $WVar$ table stores the set of global variables that have been written to by some thread during previously explored executions starting from s. These two tables are updated at the same time the global PS table is updated.

For the example in Figure 2, after exploring run 1, run 2, and run 3, we would have $WVar[n_5] = \{(x, T_1)\}$ representing that `x=20` has previously been executed by thread T_1 at some point after n_5. Similarly, we have $RVar[n_5] = \{(x, T_2)\}$ representing that `b=x` has previously been executed by thread T_2 at some point after n_5.

Whenever EXPLORE decides to skip the execution tree from a node s, we can leverage the information stored in $WVar[s]$ and $RVar[s]$ to properly update the backtrack sets for DPOR. For example, the original DPOR algorithm waits until assignment `b=x` is executed by thread T_2 before it can update the backtrack set of n_3. Now, using the entry $(x, T_2) \in RVar[n_5]$, it can put thread T_2 into the backtrack set of n_3, as if `b=x` has been executed by thread T_2 at some point after n_5.

The correctness of this solution follows Yang et al. [60] in the context of stateful DPOR, which ensure that DPOR remains sound in the presence of assertion guided pruning. For more information on the dynamic update of backtrack sets, please refer to the original description of DPOR [16].

5.3 Proof of Correctness

Now, we state and prove the correctness of our overall algorithm. Let SE_{orig} be the baseline symbolic execution procedure described in Algorithm 1, and SE_{new} be our new symbolic execution procedure with predicate summary-based pruning, as described in Algorithm 2. We say that SE_{new} is a sound reduction of SE_{orig} if it always reaches the same set of error states as SE_{orig}.

THEOREM 1. *Given a program P and an error location E. Our new symbolic execution procedure SE_{new} reaches E if and only if the original symbolic execution procedure SE_{orig} reaches E.*

Proof: We divide the proof into two steps. First, we prove that if SE_{new} reaches E, then SE_{orig} also reaches E. This is straightforward because SE_{new} explores a subset of the execution paths explored by SE_{orig}, as shown by a comparison of the two versions of NEXTSTATE in Algorithms 1 and 2.

Second, we prove that if SE_{orig} reaches E, then SE_{new} reaches E. We do this by contradiction. Assume that SE_{orig} can reach E along a path π but SE_{new} cannot. Since Lines 42–43 in Algorithm 2 are the only places where SE_{new} can skip a path, there must exist an event $\langle s, t, s' \rangle$ in path π such that $s.pcon \rightarrow \mathsf{PS}[s]$ holds under $s.\mathcal{M}$.

- Since path π is feasible, the subpath of π from s' to E must also be feasible. To skip π in SE_{new}, the subpath must have

been explored and then summarized in $\mathsf{PS}[s']$, presumably when SE_{new} first explored the subpath.
- But if $\mathsf{PS}[s']$ already includes this common subpath from s' to E, by definition, SE_{new} must have reached the error block E. This contradicts our assumption that the new symbolic execution procedure SE_{new} cannot reach the error block E.

Therefore, our assumption is incorrect. The theorem holds. $\square$

6. OPTIMIZATIONS

In our new method, the size of the summary table as well as the size of the logical constraint in each entry may become an performance bottleneck. Since large logic formulas are expensive to compute and store, we would like to reduce the associated computational cost without affecting soundness of the overall procedure. Toward this end, we propose two optimizations.

6.1 Leveraging Static Program Slicing

Our first optimization is to combine our assertion guided pruning with static program *slicing* to achieve a more significant state space reduction. Given an assertion statement st, we define the *slice* of st as the set of all statements in the program that may affect the result of st. The slice is computed based on two dependency relations: the control dependency relation and the data dependency relation. Intuitively, a statement st' is a control dependency of a statement st if the execution of st' determines whether st can be executed. Whereas a statement st'' is a data dependency of st if the execution of st'' may affect the data used in st.

```
1  if (p)
2    y = v;
3  z = w * 5;
4  if (q)
5    x = z * 2;
6    assert(x);
```

Figure 5. Example for static program slicing computation.

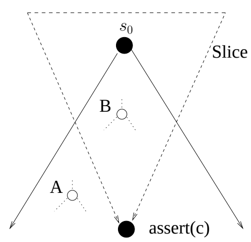


Figure 6. Using Type A and B nodes outside the slice.

Consider the example in Figure 5. The write to x at Line 5 has a *control dependency* at Line 4, and a *data dependency* at Line 3. The *slice* of Line 5 is defined as the transitive closure of its control and data dependencies, which consists of Lines 3–5. In contrast, the branching statement at Line 1 and the write to y at Line 2 are irrelevant since their execution will *not affect* the value written to x at Line 5 nor the reachability of Line 5. Therefore, for the purpose of checking the assertion at Line 6, which is related to the value of x at Line 5, we can simply ignore Lines 1–2. In other words, the slice of Line 5 (and Line 6) defines a sub-program producing an equivalent result as the full program as far as assertion checking is concerned.

We implemented the inter-procedural slicing method of Horwitz et al. [22, 40] together with an Andersen [3] style flow-insensitive alias analysis to compute the program slice statically. We implemented the method in LLVM using the Datalog engine inside the Z3 SMT solver [12]. The overall method is flow-insensitive, and safe for handling multithreaded program with sequentially consistent memory. Due to the lack of space, we do not go over the details here. Readers can refer to [27, 22, 15, 3] for more details.

We combine static program slicing with symbolic execution as follows. First, we compute the static program slice prior to the start of symbolic execution. Then, inside the symbolic execution procedure as described in Algorithm 2, for each *to-be-executed* b-PP or i-PP node s, we check if the corresponding branch condition or global operation belongs to the static slice of the assertion statement. If the answer is no, we handle a pivot point s (which can be an i-PP or a b-PP) in one of the following ways depending on the node type as illustrated in Figure 6.

- **Type A:** If s is not on any path from s_0 to the assertion statement, we treat each outgoing edge from s as if it is **halt**. In other words, we stop the current execution and backtrack from s immediately. Note that backtracking will automatically trigger the computation of weakest precondition.
- **Type B:** If s is on some GIG path from s_0 to the assertion statement, we cannot simply treat s as the end of the program since outgoing paths from s may still lead to the assertion failure. As shown in Figure 6, we have to symbolically execute at least one of the outgoing edges from the Type B node, while skipping the other outgoing edges.

The correctness of this approach directly follows from the definition of slicing. For both Type A and Type B nodes outside the program slice, *which outgoing edge to execute* does not affect the reachability of the bad state. Due to the relative efficiency of the static slicing algorithm, the overhead of computing the slice is minimal compared to the subsequent symbolic execution procedure. However, we will show through experiments that, by leveraging static program slicing results, we can significantly decrease of the number of executions to be explored, thus decreasing the complexity of the overall analysis.

6.2 Approximating the Summary Constraints

Following Theorem 1, we can prove that in general, any kind of underapproximation of $\mathsf{PS}[s]$ may be used in Algorithm 2 to replace $\mathsf{PS}[s]$, while maintaining the soundness of our pruning method. Our optimization is to heuristically reduce the computational cost associated with predicate summaries. Toward this end, we propose the following two underapproximations.

First, we use a global hash table with a fixed number N of entries to limit the storage cost for PS. With such bounded table, two global control locations s and s' may be hashed to the same entry. Whenever this happens, instead of storing both summaries in a linked list for that entry, we limit the overall cost by dropping one of them. That is, when $key(s) = key(s')$, we heuristically remove one entry, effectively setting the corresponding predicate summary false.

Second, we use a fixed threshold to bound the size of each individual logical constraint for $\mathsf{PS}[s]$. In other words, when the predicate summary becomes too large, we will stop adding new weakest-preconditions to it, thereby dropping all subsequently explored subpaths. That is,

$$\textbf{if } (\textbf{size}(\mathsf{PS}[s]) < \text{bnd}) \quad \mathsf{PS}[s] {:=} \mathsf{PS}[s] \vee wp[s] \; .$$

This is again an underapproximation of $\mathsf{PS}[s]$.

A main advantage of this on-demand constraint minimization framework is that it allows various forms of underapproximations to be plugged into it without affecting the soundness proof of the overall algorithm. With underapproximations, it is possible that we may no longer be able to prune away all redundant executions, however, we can guarantee that all pruned executions are truly redundant. In particular, the baseline symbolic execution in Algorithm 1 (no pruning) can be viewed as an extreme form of underapproximation, where $\mathsf{PS}[s]$ is underapproximated to false for all global control locations.

7. EXPERIMENTS

We have implemented our method in *Cloud9* [11], which in turn builds upon the LLVM compiler [2] and the KLEE symbolic virtual machine [8]. Note that KLEE does not by itself support multithreading, and although *Cloud9* has extended KLEE to support

Table 2. Summary of our experimental results.

			Cloud9		+DPOR		+DPOR+AG	
Name	LOC	Threads	Runs	Time (s)	Runs	Time (s)	Runs	Time (s)
fibbenchfalse1	44	2	924	61.4	48	2.0	15	1.8
fibbenchfalse2	44	2	–	>1800	628	36.2	34	3.9
fibbenchfalse3	44	2	–	>1800	8704	503.8	378	13.7
indexertrue	85	2	–	>1800	81	2.8	24	6.0
lazy01false	51	3	11	0.5	3	0.3	3	1.1
reorder2false1a	85	2	7	0.3	3	0.3	3	1.2
reorder2false1b	85	3	91	1.4	26	0.6	9	1.2
reorder2false1c	85	4	2421	89.1	205	3.2	39	1.6
reorder2false2a	85	2	23	0.6	14	0.5	14	1.5
reorder2false2b	85	3	479	8.9	233	5.0	64	2.2
sigmafalse1	49	2	12	0.4	6	0.3	2	1.2
sigmafalse2	49	3	180	3.2	50	1.0	2	1.2
sigmafalse3	49	4	4830	222.4	862	18.6	2	1.2
singletonfalse	57	4	60	1.1	24	0.6	19	1.1
stackfalse	120	2	527	8.6	236	3.9	49	2.8
stateful01true	55	2	6	0.4	6	0.4	5	1.2
twostage3false	129	3	4862	302.1	88	1.1	34	2.2
dekkertrue	55	2	–	>1800	280	3.6	6	1.5
petersontrue1	43	2	–	>1800	1052	22.7	64	2.7
petersontrue2	43	2	–	>1800	2566	86.6	85	8.1
readwritelktrue1	52	2	24	0.6	4	0.3	4	1.1
readwritelktrue2	52	4	–	>1800	–	>1800	436	14.9
timevarmutextrue	55	2	41	0.8	4	0.3	2	1.0
szymanskitrue	55	2	–	>1800	–	>1800	6	1.8
unveriftrue	40	2	–	>1800	221	2.9	27	1.7
bluetoothbad	88	2	–	>1800	1789	25.1	95	4.0
art-example	71	2	450	11.5	146	3.1	9	1.5
fsbenchbad	86	8	–	>1800	256	9.2	9	20.9
tickettrue	76	2	1062	19.6	274	4.8	44	1.9
accountbad	60	3	8	0.4	8	0.4	8	1.0
circularbufbad1	109	2	118	1.7	118	1.9	58	3.8
circularbufbad2	109	2	358	5.5	358	5.5	132	6.4
readreadwrite	50	3	96	1.4	19	0.5	3	1.1
queuefalse	167	2	252	3.9	252	3.8	26	3.9
nbds-slU1a	1942	2	–	>1800	133	8.9	5	7.8
nbds-slU1b	1942	2	–	>1800	–	>1800	76	16.2
nbds-slU1c	1942	2	–	>1800	–	>1800	202	35.2
nbds-slU2a	1942	2	–	>1800	241	25.3	29	12.8
nbds-slU2b	1942	2	–	>1800	–	>1800	118	24.5
nbds-slU2c	1942	2	–	>1800	–	>1800	717	164.8
nbds-skiplist	1994	3	–	>1800	–	>1800	1	25.1
nbds-hashw1a	2322	2	–	>1800	1339	167.4	123	177.8
nbds-hashw1b	2322	2	–	>1800	6501	1568.9	675	222.8
nbds-hashw1c	2322	2	–	>1800	–	>1800	2399	476.9
nbds-hashw2a	2234	2	–	>1800	5852	674.1	369	155.3
nbds-hashw2b	2234	2	–	>1800	–	>1800	1735	257.4
nbds-hashw2c	2234	2	–	>1800	–	>1800	4017	528.4
nbds-hash	2375	2	–	>1800	–	>1800	2283	333.8
nbds-list	1887	3	–	>1800	10274	1130.7	1	5.9
nedmalloc	6303	4	–	>1800	–	>1800	1	12.0
Average				**986.9**		**518.5**		**51.6**

a limited number of POSIX thread routines, it does not attempt to cover all feasible thread interleavings. Indeed, *Cloud9* allows for context switches only before certain POSIX thread synchronizations but not before shared variable reads/writes. Furthermore, *Cloud9* does not support partial order reduction. Instead, it forks a new execution every time a POSIX synchronization is encountered, which can cause the number of executions to explode quickly.

We have extended *Cloud9* to implement the baseline symbolic execution in Algorithm 1, which systematically explores both intra-thread paths and thread interleavings. Then, we implemented the DPOR algorithm [16]. Based on these extensions, we have implemented our new assertion guided pruning (Algorithm 2) with the optimizations presented in Section 6.

We have conducted experiments on two sets of benchmarks. The first set consists of multithreaded C programs from the 2014 Software Verification Competition (SV-COMP) benchmark [48] and programs from [14, 29]. The second set consists of two real multithreaded applications: *nbds* [35], a collection of lock-free data

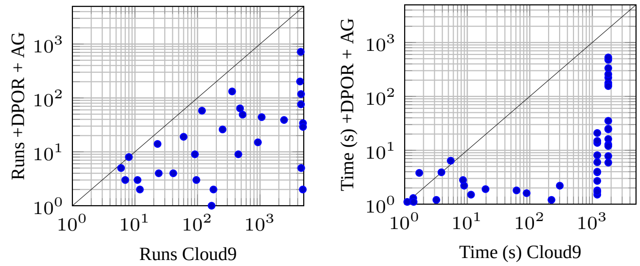


Figure 7. Scatter plots comparing our new method with Cloud9.

structures, and *nedmalloc* [36], a thread-safe malloc implementation. Each of these programs has between 40 to 6,500 lines of code, with a combined total of 40,291 lines of code. Each benchmark program is first transformed into LLVM bitcode using Clang/LLVM, before given to the symbolic execution tool with a set of user annotated variables as symbolic input.

Table 2 summarizes the results of our experimental evaluation. Columns 1–3 show the name, lines of code, and the number of threads for each program. Columns 4–9 compare the performance of three different methods in terms of the number of explored runs and the total run time in seconds. *Cloud9* denotes the baseline symbolic execution algorithm in Algorithm 1, +DPOR denotes the baseline algorithm with dynamic partial order reduction, and +DPOR + AG denotes our new method, which augments the baseline algorithm with DPOR and assertion guided pruning. The runtime of +DPOR + AG includes the time to compute the slice. For all tests, we used a maximum time of 30 minutes.

In the remainder of this section, we analyze the experimental results in more details, to answer the following research questions:

1. How effective is our proposed pruning technique? Is it more effective than DPOR alone?
2. How scalable is our technique? Is it practical in handling realistic C/C++ programs?

First, we show the comparison of Cloud9 and +DPOR + AG in two scatter plots in Figure 7, where the x-axis in each scatter plot represents the number of runs (or time) of the baseline algorithm (Cloud9), and the y-axis represents the number of runs (or time) of our method (+DPOR + AG). Each benchmark program is represented by a dot in the scatter plots; dots below the diagonal lines are winning cases for our method. The results show that our new method can significantly reduce the number of runs explored by symbolic execution as well as the overall execution time. In many cases, the baseline algorithm timed out after 30 minutes while our new method finished in a few seconds.

Next, we show the comparison of +DPOR and +DPOR + AG in the scatter plots in Figure 8. Our goal is to quantify how much of the performance improvement comes from our new assertion guided pruning as opposed to DPOR. Again, dots below the diagonal lines are winning cases for our method (+DPOR + AG) over DPOR. For most of the benchmark programs, our new method demonstrated a significant performance improvement over DPOR. For some benchmark programs, however, +DPOR + AG was slightly slower than +DPOR despite that it executed the same, or a smaller, number of runs. This is due to the additional overhead of running the supplementary static slicing algorithm, as well as predicate summary-based pruning, which did not provide sufficient performance boost to offset their overhead.

However, it is worth noting that, where our combined optimization of slicing and pruning is able to bring a performance improvement, it often leads to a drastic reduction in the execution time compared to DPOR alone. For example, in *nedmalloc* (Table 2), our new method was able to identify that the property does not depend

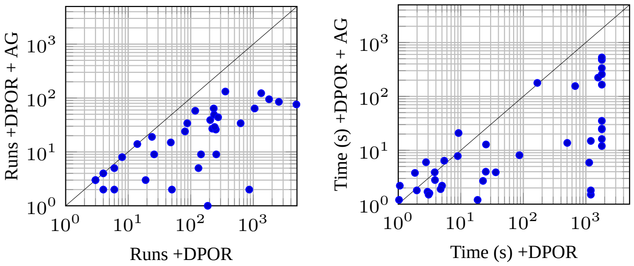


Figure 8. Scatter plots comparing our new method with DPOR.

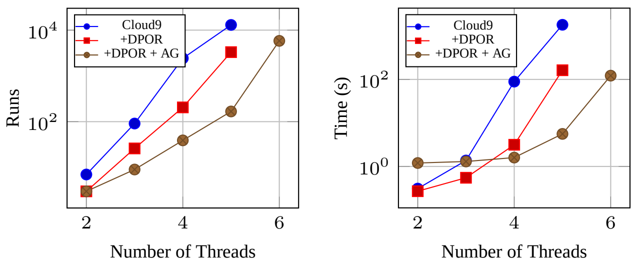


Figure 9. Parameterized results for *reorder2false* experiment.

on any shared variables. In such cases, it can safely skip exploring the entire interleaved state space and finish in just one run.

We also evaluated the growth trends of the three methods when the complexity of the benchmark program increase. Figure 9 shows the results of comparing the three methods on a parameterized program named *reorder2false*. In these two figures, the x-axis represents the number of threads created in the parameterized program, and the y-axis represents, in logarithmic scale, the number of runs explored and the execution time in seconds. As shown by these two figures, the computational overhead of all three methods increases as the complexity of the program increases. However, our new method increases at a significantly reduced rate compared to the two existing methods.

8. RELATED WORK

As we have mentioned earlier, for sequential programs, there is a large body of work on mitigating path explosion in symbolic execution, including the use of function summaries [18], may-must abstraction [20], demand-driven refinement [31], state matching [51], state merging [30], and structural coverage [37]. McMillan proposed a method called *lazy abstraction* with interpolants [33, 34], which has been shown to be effective in model checking sequential software [6]. Jaffar et al. [10] used a similar method in the context of constraint programming to compute resource-constrained shortest paths and worst-case execution time. However, a direct extension of such methods to multithreaded programs would be inefficient since they lead to the naive exploration of all thread interleavings.

Wachter et al. [52] extended McMillan's lazy abstraction with interpolants method [34] to multithreaded programs while combining it with a symbolic implementation of the monotonic partial order reduction algorithm [26, 59]. The idea is to apply interpolant-based reduction to each interleaved execution while applying symbolic POR to reduce the number of interleavings. Chu and Jaffar [9] proposed a similar method, where they improved the symbolic POR algorithm by considering not only the standard independence relation but also a new *semi-commutativity* relation. However, these existing methods [52, 9] differ from our method significantly.

First, we merge predicate summaries at interleaving pivot points whereas the existing methods [52, 9] do not. Second, we leverage static program slicing before symbolic execution and heuristic minimization of summary constraints during symbolic execution to further reduce the search space. Finally, our pruning method is designed to work seamlessly with the more scalable DPOR algorithm [16] whereas the existing methods implemented symbolic POR. Neither of these previous methods demonstrated handling C/C++ code with more than a thousand lines of code as in our work.

Kusano and Wang [29] introduced a notion of predicate dependence in the context of dynamic partial order reduction. Wang et al. [58, 53] proposed similar property-driven pruning methods for dynamic model checking. However, these methods were geared toward stateless model checking, which can be viewed as a form of systematic testing with fixed data input, as opposed to symbolic data inputs. Furthermore, these methods relied on control and data dependency relations as opposed to symbolic constraints generated from weakest precondition computation, and therefore was unable to merge non-failing executions reaching different final states. In this sense, our new method is a more general and more accurate version of these prior works. Furthermore, it is orthogonal and complementary to the symmetry-reduction method proposed by Yang et al. [61].

Our method also differs from the various heuristic state space reduction techniques [43, 38, 47] which do not guarantee the soundness of the reduction. For example, Farzan et al. [14] and Razavi et al. [39] proposed methods for exploring certain subsets of thread interleaving scenarios in symbolic execution of concurrent programs. The idea of selective thread interleaving exploration was also used by Wang et al. [57] to cover the interleaving of certain pairs of dependent operations captured by a history aware predecessor set. There are also many predictive bug detection methods based on the use of SMT solvers [55, 28, 56, 24, 25, 45, 41, 46, 45, 54], which explore only thread interleavings under fixed program inputs.

The GREEN tool by Visser et al. [50] provides a wrapper around constraint satisfiability solvers to check if the results are already available from prior invocations, and reuse the results if available. As such, they can achieve significant reuse among multiple calls to the same solvers during the symbolic execution of different paths. GREEN achieves this by distilling constraints into their essential parts and then representing them in a canonical form. The reuse achieved by GREEN is at a much lower level, and therefore is complementary to our new pruning method.

Finally, we assume the sequential consistency memory model, although it is possible to integrate our method with the dynamic partial order reduction methods for relaxed memory models [63, 1]—we leave this for future work.

9. CONCLUSIONS

We have presented a predicate summary-based pruning method for improving symbolic execution of multithreaded program. Our method is designed to work with the popular DPOR algorithm, and has the potential of achieving exponential reduction. We have implemented the method in *Cloud9* and demonstrated its effectiveness through experiments on multithreaded C/C++ benchmarks. For future work, we plan to conduct more experiments to identify the *sweet spots* in using heuristic minimizations of summary constraints to exploit the trade-off between increasing the pruning power and decreasing the computational overhead.

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. Any opinions, findings, and conclusions expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

11. REFERENCES

[1] P. A. Abdulla, S. Aronis, M. F. Atig, B. Jonsson, C. Leonardsson, and K. F. Sagonas. Stateless model checking for TSO and PSO. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 353–367, 2015.

[2] V. Adve, C. Lattner, M. Brukman, A. Shukla, and B. Gaeke. LLVM: A low-level virtual instruction set architecture. In *ACM/IEEE international symposium on Microarchitecture*, San Diego, California, Dec 2003.

[3] L. O. Andersen. Program analysis and specialization for the c programming language. Technical report, University of Copenhagen, 1994.

[4] T. Ball. A theory of predicate-complete test coverage and generation. In *Formal Methods for Components and Objects, Third International Symposium, FMCO 2004, Leiden, The Netherlands, November 2 - 5, 2004, Revised Lectures*, pages 1–22, 2004.

[5] T. Bergan, D. Grossman, and L. Ceze. Symbolic execution of multithreaded programs from arbitrary program contexts. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 491–506, 2014.

[6] D. Beyer and P. Wendler. Algorithms for software model checking: Predicate abstraction vs. impact. In *International Conference on Formal Methods in Computer-Aided Design*, pages 106–113, 2012.

[7] P. Boonstoppel, C. Cadar, and D. R. Engler. RWset: Attacking path explosion in constraint-based test generation. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 351–366, 2008.

[8] 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.

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

[10] D.-H. Chu and J. Jaffar. A complete method for symmetry reduction in safety verification. In *International Conference on Computer Aided Verification*, pages 616–633, 2012.

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

[12] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 337–340, 2008.

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

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

[15] 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.

[16] 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.

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

[18] P. Godefroid. Compositional dynamic test generation. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 47–54, 2007.

[19] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *Programming Language Design and Implementation*, pages 213–223, June 2005.

[20] P. Godefroid, A. V. Nori, S. K. Rajamani, and S. Tetali. Compositional may-must program analysis: unleashing the power of alternation. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 43–56, 2010.

[21] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *Computer Aided Verification (CAV'97)*, pages 72–83. Springer, 1997. LNCS 1254.

[22] S. Horwitz, T. W. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 35–46, 1988.

[23] 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.

[24] 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.

[25] V. Kahlon and C. Wang. Lock removal for concurrent trace programs. In *International Conference on Computer Aided Verification*, pages 227–242, 2012.

[26] V. Kahlon, C. Wang, and A. Gupta. Monotonic partial order reduction: An optimal symbolic partial order reduction technique. In *International Conference on Computer Aided Verification*, pages 398–413, 2009.

[27] K. Kennedy and J. R. Allen. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[28] 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.

[29] 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.

[30] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea. Efficient state merging in symbolic execution. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 193–204, 2012.

[31] R. Majumdar and K. Sen. Hybrid concolic testing. In *International Conference on Software Engineering*, pages 416–426, 2007.

[32] A. W. Mazurkiewicz. Trace theory. In *Advances in Petri Nets*, pages 279–324. Springer, 1986.

[33] K. L. McMillan. Lazy abstraction with interpolants. In *International Conference on Computer Aided Verification*, pages 123–136. Springer, 2006. LNCS 4144.

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

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

[36] ned productions: nedmalloc URL: http://www.nedprod.com/programs/portable/nedmalloc/.

[37] R. Pandita, T. Xie, N. Tillmann, and J. de Halleux. Guided test generation for coverage criteria. In *IEEE International Conference on Software Maintenance (ICSM 2010), September 12-18, 2010, Timisoara, Romania*, pages 1–10, 2010.

[38] S. Park, S. Lu, and Y. Zhou. CTrigger: exposing atomicity violation bugs from their hiding places. In *Architectural Support for Programming Languages and Operating Systems*, pages 25–36, 2009.

[39] N. Razavi, F. Ivancic, V. Kahlon, and A. Gupta. Concurrent test generation using concolic multi-trace analysis. In *Asian Symposium on Programming Languages and Systems*, pages 239–255, 2012.

[40] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 49–61, New York, NY, USA, 1995. ACM.

[41] M. Said, C. Wang, Z. Yang, and K. Sakallah. Generating data race witnesses by an SMT-based analysis. In *NASA Formal Methods*, pages 313–327, 2011.

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

[43] K. Sen. Race directed random testing of concurrent programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 11–21, 2008.

[44] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 263–272, 2005.

[45] A. Sinha, S. Malik, C. Wang, and A. Gupta. Predicting serializability violations: Smt-based search vs. dpor-based search. In *Haifa Verification Conference*, pages 95–114, 2011.

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

[47] F. Sorrentino, A. Farzan, and P. Madhusudan. PENELOPE: weaving threads to expose atomicity violations. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 37–46, 2010.

[48] SV-COMP. 2014 software verification competition. URL: http://sv-comp.sosy-lab.org/2014/, 2014.

[49] N. Tillmann and J. de Halleux. PEX – white box test generation for .NET. In *International Conference on Tests and Proofs*, pages 134–153, 2008.

[50] W. Visser, J. Geldenhuys, and M. B. Dwyer. Green: reducing, reusing and recycling constraints in program analysis. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, page 58, 2012.

[51] W. Visser, C. S. Pasareanu, and R. Pelánek. Test input generation for java containers using state matching. In *International Symposium on Software Testing and Analysis*, pages 37–48, 2006.

[52] B. Wachter, D. Kroening, and J. Ouaknine. Verifying multi-threaded software with Impact. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 210–217, 2013.

[53] 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.

[54] C. Wang and M. Ganai. Predicting concurrency failures in generalized traces of x86 executables. In *International Conference on Runtime Verification*, pages 4–18, Sept. 2011.

[55] 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.

[56] C. Wang, R. Limaye, M. Ganai, and A. Gupta. Trace-based symbolic analysis for atomicity violations. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 328–342, 2010.

[57] C. Wang, M. Said, and A. Gupta. Coverage guided systematic concurrency testing. In *International Conference on Software Engineering*, pages 221–230, 2011.

[58] C. Wang, Y. Yang, A. Gupta, and G. Gopalakrishnan. Dynamic model checking with property driven pruning to detect race conditions. In *International Symposium on Automated Technology for Verification and Analysis*, pages 126–140, 2008.

[59] 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.

[60] Y. Yang, X. Chen, G. Gopalakrishnan, and R. Kirby. Efficient stateful dynamic partial order reduction. In *SPIN Workshop on Model Checking Software*, pages 288–305, 2008.

[61] Y. Yang, X. Chen, G. Gopalakrishnan, and C. Wang. Automatic discovery of transition symmetry in multithreaded programs using dynamic analysis. In *International SPIN workshop on Model Checking Software*, pages 279–295, 2009.

[62] 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.

[63] N. Zhang, M. Kusano, and C. Wang. Dynamic partial order reduction for relaxed memory models. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2015.