ConTesa: Directed Test Suite Augmentation for Concurrent Software

Tingting Yu, Zunchen Huang, and Chao Wang

Abstract—As software evolves, test suite augmentation techniques may be used to identify which part of the program needs to be tested due to code changes and how to generate these new test cases for regression testing. However, existing techniques focus exclusively on sequential software, without considering concurrent software in which multiple threads may interleave with each other during the execution and thus lead to a combinatorial explosion. To fill the gap, we propose *ConTesa*, the first test suite augmentation tool for concurrent software. The goal is to generate new test cases capable of exercising both code changes and the thread interleavings affected by these code changes. At the center of *ConTesa* is a two-pronged approach. First, it judiciously reuses the current test inputs while amplifying their interleaving coverage using random thread schedules. Then, it leverages an *incremental symbolic execution* technique to generate more test inputs and interleavings, to cover the new concurrency-related program behaviors. We have implemented *ConTesa* and evaluated it on a set of real-world multithreaded Linux applications. Our results show that it can achieve a significantly high interleaving coverage and reveal more bugs than state-of-the-art testing techniques.

1 Introduction

Regression testing is a widely-used technique to re-validate evolving software. Typically, engineers begin by executing the existing test cases, to which various techniques for *test selection* [28], [42], [47], [48], [51] and *test case prioritization* [15], [16], [32], [35], [49], [55], [66] may be used, to reduce the testing cost. However, existing test cases may not be sufficient for covering the new or modified software code and related program behaviors. To address this problem, *regression test suite augmentation (RTA)* is used to identify where new test cases are needed and then create these test cases [8], [50], [56], [65]. However, prior work on RTA focuses exclusively on sequential software; it does not consider issues related to thread interleavings or have the ability to effectively cover concurrency-related new program behaviors.

Unlike sequential software, for which RTA has to consider the addition of test inputs only, concurrent software requires RTA to generate both test inputs and thread interleavings so as to cover the new concurrency behaviors of the modified code. This is often difficult because in real-world applications, the thread interleaving space may be too large to explore exhaustively [36]. Although there is a large body of work on mitigating this *interleaving explosion* problem, including techniques based on static analysis [18], [29], [33], [39], [62], [64], systematic exploration [4], [11], [38], [52], [53], [61], [63], and schedule generation [41], [46], they all target a single program version and thus do not directly benefit RTA. In RTA, one must consider two closely-related program versions. Terragni et al. [57] have proposed a technique in the context of regression testing for exploring alternative thread interleavings pertinent to the affected shared-variable (SV) accesses, and showed that only 1% of SV accesses in real applications are affected. However, their technique relies exclusively on the existing test inputs; it cannot generate new test inputs.

In this paper, we present *ConTesa*, the first RTA technique for concurrent software to simultaneously generate new test inputs and thread interleaving schedules. To leverage the existing test inputs while minimizing the cost of generating new test inputs, *ConTesa* uses a two-pronged approach. In the first phase, *ConTesa* takes the two program versions P and P' and computes a set IS_Δ of interleaving schedules that need to be covered. To compute these *coverage targets*, we consider the global operations (e.g., shared-variable accesses and synchronizations) affected by code changes and select interleavings to cover each of them at least once. Since exhaustively covering all interleavings is practically infeasible, we use a modest coverage criterion: the set of selected interleavings must be able to cover a predefined set of inter-thread *definition-use (DU)* access patterns.

In the second phase, *ConTesa* generates new test inputs to activate the predefined set of DU pairs. Prior to doing that, however, *ConTesa* utilizes the existing test inputs together with random schedules to quickly trim down the coverage targets in IS_Δ. Thus, new test inputs are generated only for coverage targets that cannot be reached by existing test inputs. For each coverage target left in IS_Δ, i.e., a DU pair, *ConTesa* uses an SMT solver-based symbolic execution procedure to compute the new test input, as well as the thread schedule under which the coverage of program version P' can be increased. This iterative process of generating new test inputs and thread schedules repeats until the entire set IS_Δ is covered or a predefined time limit is reached.

To evaluate *ConTesa*, we conducted experiments on the regression testing of 13 real-world C/C++ applications, including four large programs with 95K to 154K lines of code. We compare *ConTesa* with three state-of-the-art techniques for testing multithreaded software. The first technique is Conc-ise [21], an incremental symbolic execution tool for exploring the affected interleaving space. Conc-ise does not reuse existing test cases to guide the exploration; in contrast, it generates all test inputs and thread interleavings from scratch. The second technique is Con2colic [17], a technique for systematically generating test inputs and thread schedules via symbolic execution of the new program version. The third technique is ReConTest [57], a regression testing tool that explores the interleaving space using only existing test inputs. More details on comparing *ConTesa* to these three baseline techniques are discussed in Section 2.3.

Our results show *ConTesa* outperforms all three techniques in terms of the test coverage, fault detection rate, and testing time. Specifically, compared to the first approach, *ConTesa* detected 18.1% more faults, improved the coverage by 4.5%, and was up to 1.9 times faster; compared to the second approach, *ConTesa* detected 44.4% more faults, improved the coverage by 4.9%, and was up to 39.3 times faster; compared to the third approach, *ConTesa* was three times slower but detected 85.7% more faults and improved the coverage by 52.5%.

One key benefit of *ConTesa* is the reuse of existing test inputs together with random schedules. This is because existing test cases provide a rich source of data on potential inputs and code reachability, which can speed up the exploration of the affected program space. Compared to the three baseline approaches, *ConTesa* performs better especially on larger programs. Moreover, existing test cases are naturally available as a starting point in the regression testing context. There are dynamic analysis techniques for detecting concurrency faults that utilize existing test cases, which can be combined with *ConTesa* to further improve fault detection effectiveness.

In summary, this paper makes the following contributions:

- We propose the first regression test augmentation (RTA) tool for concurrent software, capable of utilizing existing test cases as well as generating new test cases (inputs and thread schedules).
- We conduct controlled experiments on real applications to evaluate different input and interleaving generation strategies and demonstrate the effectiveness of the proposed technique.

In the remainder of this paper, we first introduce the technical background and problem statement together with a motivating example in Section 2. Then, we present the overall algorithm of *ConTesa* in Section 3, followed by detailed descriptions of the test augmentation algorithm in Section 4. We present the empirical study and results in Sections 5 and 6, respectively, followed by a discussion of our observations in Section 7. We present the related work in Section 8. Finally, we give our conclusions in Section 9.

2 Background and Motivation

We first define the *test suite augmentation* problem and then illustrate the technical challenges in the context of concurrent software using a motivating example.

2.1 Test Suite Augmentation

Let P be a program, P' be a modified version of P, T be the existing test suite[1] for P, and T' be the new test suite for P'. Our goal is to compute T' when given P, P', and T.

1. A test suite is a collection of test inputs.

```
Thr1 :                                  Thr2 :

  1.if (y < 0) {                        15. mutex_lock(&lk);
  2.   mutex_lock(&lk);                 16. x = y - 1;
  3.   if (x > 3) {                     17. mutex_unlock(&lk);
       ...                                      ...
++4.   mutex_unlock(&lk);               18. mutex_lock(&lk);
  5.      y = x + 1;                    19. if (x < 3 && y == -1)
  6.      assert (y != 0);              20.   y = y + 1;
  7.     }                              21. mutex_unlock(&lk);
  8.  else {
      ...
++9.   mutex_unlock(&lk);
  10.  }
--11.  mutex_unlock(&lk);
  12. }
  13.if ( z < 3) {
    ...
  14. }
```

Fig. 1: A program with deleted ("--") and added ("++") lines.

In regression testing, engineers often begin by reusing T. Since reusing all test cases (i.e., the *retest-all approach*) is expensive, *regression test selection (RTS)* attempts to select, from T, a subset $T' \subseteq T$ of test cases that are important, while omitting test cases that are not as important [28], [42], [47], [48], [51]. Similarly, *regression test prioritization (RTP)* attempts to reorder the existing test cases in T with the goal of more quickly reaching the testing objectives. The most obvious testing objective is revealing the faults [15], [16], [32], [35], [49], [55], [66].

Clearly, both RTS and RTP are concerned with the reuse of test cases in T. In contrast, *regression test augmentation (RTA)*, which is the focus of this paper, is concerned with 1) identifying the affected entities in the software code (e.g., portions of P' or its specification for which new test cases are needed), 2) checking whether existing test suites are adequate for covering the affected entities , and 3) creating *new* test cases to exercise these affected but not-yet-covered entities [8], [50], [56], [65].

At the center of RTS, RTP, as well as RTA is a static program analysis named *change-impact analysis (CIA)*. It is used to analyze the program models (e.g., control-flow graphs) of both P and P' and determine the code changes as well as program entities affected by these code changes. Thus, CIA serves as a foundation for performing the various steps of regression testing. In principle, only program entities that are involved in the code changes or affected by these changes need to be re-tested.

2.2 A Motivating Example

We use the concurrent program in Fig. 1 to illustrate the main challenges in RTA. The program is a simplified and slightly modified version of the code snippet from `Aget`. The program has two threads Thr_1 and Thr_2 as well as global variables `x`, `y`, and `z`. Both the original program P and the modified program P' are shown in the figure, where deleted and added lines are denoted by "--" and "++", respectively. The three lines of code changes from P to P' introduce a concurrency bug: since the execution of Lines 5-6 is no longer *atomic*, the value of `y` written at Line 5 may be modified by the other thread via the write operation at Line 20, leading to an assertion failure. Since this is a newly added program behavior – it exists in P' but not in P –

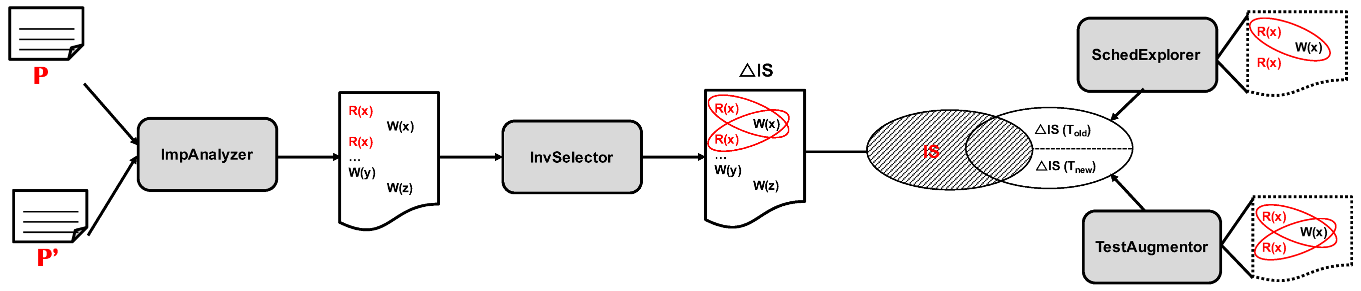

Fig. 2: Overall flow of *ConTesa*, the first regression test suite augmentation (RTA) framework for concurrent software.

new test cases capable of covering it must be generated and added to the existing test suite.

We now use the example to illustrate in detail how *ConTesa* is able to generate new tests to expose this concurrency failure. Details of the algorithms are described in Section 3 and Section 4. First, *ConTesa* employs a change-impact analysis (CIA) to identify concurrency elements (e.g., shared variables) involved in or affected by the three lines of code changes (Section 3.1). Specifically, we consider global variables x and y at Lines 5-6 as *affected*, because they are no longer protected by the lock lk. These two affected global variables can result in new thread interleavings, i.e., interleavings that are feasible in P' but infeasible in P. In contrast to x and y, we consider variable z at Line 13 as *not-affected* because the change does not affect how thread Thr_1 interacts with thread Thr_2.

Next, *ConTesa* constructs the interleaving schedule targets from the impacted concurrency elements (Section 3.2). These targets are constructed based on a practical concurrency coverage criterion: the definition-use (DU) pairs of shared variables [36]. That is, *ConTesa* identifies the set of inter-thread DU pairs such that, for the two SV accesses in each pair, at least one of them is affected by the code changes. Intuitively, a DU pair describes the change propagation across different threads. For the example program in Fig. 1, the DU pairs are $dup_{\texttt{aff}_1} = <$ (5, $w(y)$), (16, $r(y)$) $>$, $dup_{\texttt{aff}_2} = <$ (16, $w(x)$), (5, $r(x)$) $>$, $dup_{\texttt{aff}_3} = <$ (5, $w(y)$), (20, $r(y)$) $>$, $dup_{\texttt{aff}_4} = <$ (20, $w(y)$), (6, $r(y)$) $>$, where for each access, the number denotes the corresponding line in the code, w denotes a write operation, and r denotes a read operation.

Suppose $t_1 \in T$ is an existing test input such that t_1 = {x = 0, y = -2, z = 0}. Prior to generating new test cases, *ConTesa* computes the set of DU pairs that can be covered by executing the existing test input together with some random thread schedules. Specifically, for t_1, if it is executed on P' under the following thread schedule σ_1: 1→2→3→9→15→16→17→18→19→21, none of the four DU pairs would be covered. In this case, *ConTesa* needs to generate new test inputs (and their corresponding thread schedules) to cover these four DU pairs (Section 4).

To cover $dup_{\texttt{aff}_1}$, *ConTesa* uses symbolic execution along the path explored by t_1 to generate the new test input t_2 = {x = 5, y = -5, z = 0} and the schedule σ_2: 1→2→3→4→5→6→15→16→17→18→19→21. In other words, when using t_2 together with σ_2, *ConTesa* can cover the target $dup_{\texttt{aff}_1}$. To cover $dup_{\texttt{aff}_2}$, *ConTesa* does not need to generate another test input. Instead, it simply explores an alternative thread schedule under the same test input t_2 and see if it covers the other DU pairs (Section 4.1). Since running the program under t_2 and σ_2 already covers the SV read access (5, $r(x)$) followed by the write access (16, $w(x)$), *ConTesa* flips the execution order of the two events to cover $dup_{\texttt{aff}_2}$.

At this moment, the two remaining targets are $dup_{\texttt{aff}_3}$ and $dup_{\texttt{aff}_4}$. *ConTesa* again invokes symbolic execution to generate t_3 = {x = 5, y = -1, z = 0} and the schedule: σ_3: 1→2→3→4→5→15→16→17→18→19→20→21→6, which covers the target $dup_{\texttt{aff}_3}$. Then, it seeks an alternative thread schedule by flipping the order of events (20, $w(y)$) and (6, $r(y)$) to cover $dup_{\texttt{aff}_4}$. Under this particular test input and thread schedule, the assertion failure is revealed.

2.3 Relation to Existing Techniques

Since the assertion failure in Fig. 1 requires a combination of test input and thread schedule to manifest, none of the existing regression testing techniques, such as SIMRT [69], CAPP [26], or RECONTEST [57], is effective. For example, SIMRT only performs test input selection but does not consider the impact of code changes in the thread interleaving space. CAPP explores the change-induced thread interleaving space but only for the given test inputs. RECONTEST improves over CAPP by focusing the search on a small set of problematic interleavings whose shared variables are affected by code changes. However, all of these tools rely on existing test inputs and therefore are restricted to exploring a subset of affected thread interleavings. In contrast, *ConTesa* not only explores the affected interleavings under existing test inputs but also generates entirely new test inputs. As such, it has the capability of driving the search through program paths and thread interleavings not covered by these prior techniques, which in turn lead to the detection of more bugs.

At the same time, *ConTesa* is significantly more efficient than test case generation techniques that focus only on the new program P' while completely ignoring the original program P. For example, while symbolic execution based testing techniques for concurrent programs [17], [22], [54] can generate new test inputs and thread interleaving schedules, they focus on a single program version (P') and thus do not leverage the knowledge of what changed in P' compared to P or the existing test suite T for P. Consider our running example in Fig. 1: if CON2COLIC [17] were to be applied to P', it would have unnecessarily explored the unaffected interleavings related to variables in the else-branch of Thr_1 at Line 8, as well as the unaffected variable z at Line 13.

Although CONC-ISE [21] can leverage two program versions P and P' to reduce the cost of exploring both program path and thread schedules in concurrent programs, it does not reuse existing test cases in T to guide the symbolic execution or schedule exploration. For the example of Fig. 1, CONC-ISE would invoke symbolic execution four times to exercise the affected DU pairs. In contrast, a main advantage of *ConTesa* is to leverage the existing test cases in T (for P) to speed up the testing of P'.

3 OVERALL APPROACH

Fig. 2 shows the overall flow of *ConTesa*, which consists of four main components: `ImpAnalyzer` for conducting change-impact analysis, `InvSelector` for selecting interleaving targets, `SchedExplorer` for exploring thread schedules, and `TestAugmentor` for generating test inputs.

Given the set C' of code changes made to the original program P to produce the new program P', `ImpAnalyzer` first computes a list L'_{SV} of program locations involving shared variable accesses. It then identifies a subset $L'_{SV_{\texttt{aff}}} \subseteq L'_{SV}$ of these variables that may be affected by C'. *ConTesa* focuses on identifying the affected shared variables because the triggering of concurrency bugs considered in this work depends on the exposure of erroneous inter-thread memory dependencies. Next, `InvSelector` iterates through $L'_{SV_{\texttt{aff}}}$ and identifies the shared variables that match a given access pattern PT_{inv}. By default, our access pattern is the DU pair, meaning `InvSelector` produces a list of impacted DU pairs, which form the interleaving schedule targets in IS_{Δ}.

Next, *ConTesa* generates either new thread schedules or new test inputs to cover the DU pairs in IS_{Δ}. To avoid repetition, it first applies each existing test input together with a random schedule, and then invokes `SchedExplorer` to seek alternative interleavings. At the end of this process, if some DU pairs in IS_{Δ} have not been covered, it invokes `TestAugmentor`.

Inside `TestAugmentor`, *ConTesa* first executes the program P' using an existing test input that reaches the branches of the affected SV in an not-yet-covered DU pair. Then, it performs symbolic execution of all local paths (with a loop bound 10) within the threads, as well as the global path across threads. Finally, it uses the path and thread constraints gathered along the execution to generate a new data input and a new interleaving schedule, to drive the execution along a different path on a subsequent iteration (this is accomplished by negating a predicate in the path condition constraint during symbolic execution).

In general, `SchedExplorer` and `TestAugmentor` must be applied iteratively until all DU pairs in IS_{Δ} are covered, or a predefined time limit is reached.

In the subsequent sections, we describe each of the four components in more detail.

3.1 Concurrent Change-Impact Analysis

Identifying program entities affected by changes is a key step in *ConTesa*. `ImpAnalyzer` is designed to conduct this change-impact analysis. To handle concurrent software, `ImpAnalyzer` extends the change-impact analysis procedure of SimRT [69] by considering not only *standard* synchronizations (e.g., locks) but also *ad-hoc* synchronizations (e.g., busy-waiting over loops using flags). In SimRT, a concurrency element (e.g., shared variable) is regarded as *impacted* only when it is changed, or all standard synchronizations it depends on are changed. However, this approach cannot detect elements affected by thread-local changes or changes involving ad-hoc synchronizations. *ConTesa* addresses both issues.

Specifically, `ImpAnalyzer` first produces a list L'_{SV} of variables that may be accessed by multiple threads in P'. These variables are computed using the shared variable detection algorithm of Kahlon et al. [31], where each access is labeled as either "write (def)" or "read (use)" through our analysis. Next, `ImpAnalyzer` takes the program P' and L'_{SV} to compute a list $L'_{SV_{\texttt{aff}}} \subseteq L'_{SV}$ of affected shared variables in two steps.

In the first step, `ImpAnalyzer` computes a change set (i.e., a set of changed instructions), denoted by Δ_{diff}, using a lightweight *diff* utility. Since the results reported by the standard *diff* command in Linux may generate too many false positives (e.g., changing a variable name from x to y would cause all lines referring to x as changed even if they are structurally the same), we build the abstract syntax trees (ASTs) of both P and P' and compare them structurally: we traverse them in parallel to collect type and name mappings. Thus, two variables are considered equal if we encounter them in the same syntactic position reported by the *diff* tool. The ultimate changes are added to the change set C.

In the second step, `ImpAnalyzer` computes, for each changed instruction $c \in C$, the shared variables affected by c and adds them into $L'_{SV_{\texttt{aff}}}$. Thus, we perform an intra-thread forward program slicing to identify all instructions, I, that depend on c, and add the shared variables in I to $L'_{SV_{\texttt{aff}}}$ because they may affect the interleaving space of P'. Note that $I \cup c$ may contain local variables, shared variables, as well as synchronizations. Next, for each synchronization $s \in I \cup c$, we compute all shared variables protected by s, and add them to $L'_{SV_{\texttt{aff}}}$.

If s is a mutex operation, we record its synchronization context (i.e., locksets used to protect the shared variables). If the locksets on the same SV are different across two program versions, the SV is affected and thus added to $L'_{SV_{\texttt{aff}}}$. In our static analysis, the lockset of a shared variable SV is the set of locks that may be held along any path leading to the access of SV. We use the context-sensitive lockset analysis in [30] to compute the lockset for each SV access. Specifically, we construct a concurrent control flow graph (CCFG) [30] and uses backward slicing to capture all locks that may reach the access of the shared variable. The lock sets computed by our static analysis may induce false positives due to infeasible paths, which fortunately is eliminated by the subsequent dynamic phase of the test augmentation. For the program in Fig. 1, the lockset for the variables (`x` and `y`) at Lines 5-6 are changed from $\{lk\}$ to $\{\phi\}$, so these variables are affected and thus added to $L'_{SV_{\texttt{aff}}}$.

If s is a wait-signal synchronization, all SVs that are control-dependent on the `wait` operation and all SVs that can reach the `signal` operation are affected and thus added to $L'_{SV_{\texttt{aff}}}$.

3.2 Identifying the Impacted Interleaving Space

The impacted interleaving space, $IS_{\triangle}$, is computed inside `InvSelector` by selecting thread schedules containing at least one affected variable in $L'_{SV_{\text{aff}}}$. Since the number of thread interleavings may grow exponentially as the length of the execution increases, we adopt a more practical coverage criterion. The goal is to help select a small number of representative thread interleavings.

The current implementation of *ConTesa* employs an inter-thread def-use (DU) criterion [36], which is satisfied if and only if a write `w` in one thread happens before a read `r` in another thread and there is no other write to the same variable between them. Each DU pair containing at least one affected SV is considered impacted and thus added to $IS_{\triangle}$. During test augmentation, we leverage new symbolic execution algorithms to efficiently generate test inputs as well as thread schedules to cover all the DU pairs in $IS_{\triangle}$.

4 The Main Test Augmentation Algorithm

The test augmentation procedure is shown in Algorithm 1, which contains two subroutines `SchedExplorer` and `TestAugmentor`. It begins with an initial set T of test inputs and a set $IS_{\triangle}$ of DU pairs serving as the coverage target TG (line 1). During the iterative process, we first invoke `SchedExplorer` to execute all test inputs in T with random thread schedules and check if any DU pair $dup \in TG$ is covered (Line 4). *ConTesa* could also be used in conjunction with SimRT [69] or Recontest [58] to speed up the exploration. For each remaining DU pair that is not covered, we invoke `TestAugmentor` to generate a new input t_{new} together with a thread schedule s_{new} (Line 6) capable of covering dup. If t_{new} and s_{new} are successfully computed and executed on P', the corresponding DU pair is covered and t_{new} is added to the new test suite T_{new} (Lines 7-10).

The order in which events in $IS_{\triangle}$ are considered may affect the performance of *ConTesa*. Here, $SV_{\triangle}$ denotes all shared variables in $IS_{\triangle}$ and each shared variable in $SV_{\triangle}$ is a single coverage target. We have investigated the use of a breadth-first search (BFS) order of shared variables in $SV_{\triangle}$. Our conjecture is that, by guiding *ConTesa* toward paths that explore the most *not-yet-covered* targets in the BFS order may speed up the augmentation process, because test cases generated earlier in process may cover shared variables occurring later in process, thus obviating the need to consider them again. It may also maximize the number of interleavings to be explored per test input, thus reducing the symbolic execution cost for generating more test inputs – see our detailed discussions in Section 7.

The manner in which test inputs are used also affects the performance of *ConTesa*. In this work, we allow for the possibility of adding newly generated test inputs back into our set of available test inputs. Specifically, if the Boolean flag $UseNew$ is set to true, the procedure in Algorithm 1 will combine newly generated test inputs with original test inputs (Lines 13-15), and this combined T will be used for the next iteration of our algorithm. By default, $UseNew$ is set to true. We shall discuss the impact of this option in more detail in Section 7.

Algorithm 1 *ConTesa* main augmentation algorithm

```
Input: $P', T, IS_{\triangle}$
Output: $T_{new}$
 1: $TG = IS_{\triangle}$
 2: while $TG \neq \phi$ and $Time \leq Limit$ do
 3:     for each $dup \in TG$ do
 4:         SchedExplorer($P', T, dup$)
 5:         if $dup$ is not covered then
 6:             <$t_{new}, s_{new}$> = TestAugmentor($P', T, dup$)
 7:             $success$ = Execute($P', t_{new}, s_{new}$)
 8:             if $success$ is $true$ then
 9:                 $T_{new} = T_{new} \cup t_{new}$
10:             end if
11:         end if
12:         update($TG$)
13:         if $UseNew$ then
14:             $T = T_{new} \cup T$
15:         end if
16:     end for
17: end while
```

4.1 Discovering New Interleaving Space

ConTesa invokes `SchedExplorer` to explore the interleaving space affected by code changes using existing test inputs, as shown at Line 4 of Algorithm 1. If a coverage target can be reached, there is no need to generate new test inputs. Here, `SchedExplorer` exercises each test input $t \in T$ on P' with a random schedule, which leads to a concrete execution $TR(t)$. If a target DU pair dup is exercised, TG is updated (Line 12) to reflect that dup is covered by $TR(t)$.

If a dup is not covered but its flipped pair (<read, write>) is exercised by $TR(t)$, we try to generate alternative interleavings to cover dup. Toward this end, we construct a partial order graph (POG) [24], which captures the ordering of concurrent events in $TR(t)$. Let (V, E) be the POG, where V is the set of nodes corresponding to the events in $TR(t)$ and E is the set of edges between these nodes. Each edge $(e_i, e_j) \in E$ represents a must-happen-before order between e_i and e_j. To generate new thread schedules, we iteratively pick a pair $E_p = (R_m, W_m)$, where R_m is a read and W_m is a write with respect to the same memory location m. If $(W_m, R_m) \in dup$, we try to generate a new interleaving by flipping the two events in E_p while respecting the other ordering constraints in the POG. In other words, derives the position of the legal flip by analyzing the POG.

To ensure the new thread schedule is feasible, we replay it on P'. If we can execute the program P' under the test input and new thread schedule, we update TG to record that the particular dup is covered. However, if such a thread schedule cannot be generated, e.g., due to violations of some ordering constraints in the POG or the infeasibility of the generated schedule, the dup is not-yet-covered.

It is also possible that executing the new thread schedule leads to deadlock when the POG does not precisely model all synchronization primitives (e.g., some ad hoc synchronizations written using shared variables). To address this problem, ConTesa sets a timeout value (i.e., 30 seconds) just in case that a deadlock occurs.

For the example program in Fig. 1, if the to-be-covered DU pair is <(16, w(x)), (3, r(x))>, but the related order <(3, r(x)), (16, w(x))> has been exercised by the execution trace $TR(t)$, we try to generate an alternative thread schedule by flipping the order of events (3, r(x)) and (16, w(x)). What is interesting is that there can be multiple ways to execute (3, r(x)) after (16, w(x)), some of which are better than the

Algorithm 2 *ConTesa* input augmentation: `TestAugmentor`

```
Input: $P'$, $T$, an uncovered DU pair $dup$
Output: $NT$
 1: $T_{cur}$ = $T$ // a set of the current test inputs
 2: $NT$ = $\phi$ // a set of all new test inputs generated
 3: while $NT_{cur} \neq \phi$ do
 4:     $NT_{cur}$ = $\phi$ //a set of newly generated test inputs in the
        current execution of line 3 to line 25
 5:     if $dup$ contains at least one uncovered $sv$ then
 6:         $T_{dup}$ = {all test inputs in $T_{cur}$ that reach $\overline{b_{dup.w}}$ or $\overline{b_{dup.r}}$}
 7:     else if isFlip($dup$) is false then
 8:         $T_{dup}$ = {all test inputs in $T_{cur}$ that reach the unflipped pair}
 9:     end if
10:     $TR_{sym}$ = {all symbolic traces obtained from executing $T_{dup}$}
11:     for each $st \in TR_{sym}$ do
12:         $\phi_{pc_w}$ = DelPred ($pc_{dup.w}$, $neg$)
13:         $\phi_{pc_r}$ = DelPred ($pc_{dup.r}$, $neg$)
14:         $\Phi = \phi_{pc_w} \wedge \phi_{pc_r} \wedge \phi_{sync} \wedge \phi_{rw}$
15:         if $\Phi$ is not seen before then
16:             $<t_{new}, s_{new}>$ = Solve ($\Phi$)
17:         end if
18:         if $t_{new} \neq$ UNSAT and $<t_{new}, s_{new}>$ is replayed then
19:             $NT_{cur}$ = $NT_{cur} \cup t_{new}$
20:         end if
21:     end for
22:     $T_{cur}$ = $NT_{cur}$
23:     $NT$ = $NT \cup NT_{cur}$
24: end while
25: return $NT$
```

others. For example, if we try to execute (3, r(x)) immediately after (16, w(x)), we would violate the mutual exclusion constraints imposed by mutex locks. Therefore, during the reordering of events, our algorithm has to consider not only events involved in the target dup but also events transitively depending on these events. For the example program in Fig. 1, it means we need to move both events (2, lock) and (3, r(x)) after (17, unlock).

4.2 Test Suite Augmentation

When a $dup \in TG$ cannot be covered by interleavings of any existing test input, we generate a new test input using symbolic execution. Algorithm 2 shows the test input generation procedure, which accepts the following parameters as input: the modified program P', the set T of existing test inputs, and a to-be-covered DU pair (dup), and returns a set NT of new test inputs as output. Furthermore, each new test input is associated with a thread schedule for covering a previously unreachable DU pair.

We use the to-be-covered dup to guide the computation of per-thread path condition constraints and inter-thread ordering constraints. Let b_{sv} denote the branch that covers a shared-variable access (sv) and $\overline{b_{sv}}$ denote the alternative branch; that is, if b_{sv} is a then-branch, $\overline{b_{sv}}$ is the corresponding else-branch. Note that b_{sv} does not have to be the immediate dominator block of sv. In `if(c1){if(c2){read(sv);}}`, for instance, the then-branches of both $c1$ and $c2$ can be considered as b_{sv} for the shared variable access sv.

Initially, the set T_{cur} is set to be T (Line 1) and NT is an empty set (Line 2). The procedure starts by reseting the newly generated input set NT_{cur} (Line 4). It then selects test inputs from T_{cur} that can 1) reach $\overline{b_{sv}}$, where sv is either a read ($du.r$) or a write ($du.w$) in the dup; and 2) reach both events in the dup that cannot be flipped by `SchedExplorer` (Line 7). If such a test input does not yet exist in T_{cur}, we use symbolic execution to generate a new test input by solving the corresponding symbolic constraints (Lines 11–21).

Our symbolic execution encodes both thread-local path constraints and inter-thread synchronization and memory constraints. Specifically, our procedure first computes the path conditions (pc_w and pc_r) to reach the branches that execute both events in the dup (Lines 12-13). There are three possible cases: 1) neither $dup.w$ nor $dup.r$ is covered, 2) only one event ($dup.w$ or $dup.r$) is covered, and 3) both $dup.w$ and $dup.r$ are covered but their order cannot be flipped by `SchedExplorer`. Inside `DelNeg(`pc`, `neg`)`, if an event is covered, neg is set to false; otherwise, neg is set to true. Inside `DelNeg`, if neg is true, a new pc is generated by negating the current branch in pc and removing all subsequent branches. If neg is false, the new pc is generated by removing all subsequent branches without negating the branch. For example, `DelNeg(`$b_1 \wedge b_2 \wedge b_3$`, `$true$`)` $= b_1 \wedge b_2$.

In addition to computing the path conditions (pc_w and pc_r), we obtain ordering constraints for synchronizations and shared memory accesses (ϕ_{sync} and ϕ_{rw}), which form the global ordering constraint Φ. During each iteration, if the constraint Φ has not been decided before, we invoke an SMT solver to decide it (Line 15). If Φ is satisfiable and the solver returns a solution $<t_{new}, s_{new}>$, which represents a new test input and the corresponding thread schedule, we add t_{new} to the set NT_{cur} of newly generated test inputs.

At the high level, the algorithm iterates through all path conditions whose execution traces reach the target branch (Line 11). This allows it to generate more test inputs and reach predicates following the shared-variable access sv, which may control additional branches that need to be covered.

Next, we describe two key steps of the symbolic execution procedure: symbolic trace collection and constraint modeling.

4.2.1 Symbolic Trace Collection

Given a test input t, *ConTesa* first executes P' under an arbitrary thread schedule to produce an execution trace. As the symbolic execution proceeds, *ConTesa* records information about the control flow, thread synchronization, and shared memory accesses within each thread. Thus, a symbolic trace contains the following three types of information.

First, the trace includes a path condition per thread, which is the sequence of control-flow decisions made in the thread. Second, the trace includes inter-thread synchronizations such as *lock*, *unlock*, *wait*, *signal*, *thread fork*, and *thread join*, together with the synchronization objects (e.g., mutex locks). Third, the trace includes the read and write accesses to shared memory. Since the value returned by a global read depends on the interleaving schedule, we represent it as a fresh symbolic value. On the other hand, the value written to a shared-variable may be either symbolic or concrete.

Table 1 shows an example, where Column 1 represents a concrete execution of the program in Fig. 1 under $t1$ = {x = 0, y = -2, z = 0}. Here, we use $W^{t.i}v$ to denote the value written to v at Line i of thread t, $R^{t.i}v$ to denote the value read from v at Line i of thread t, $L^{t.i}o$ to denote the acquisition of lock o at Line i of thread t, and $U^{t.i}o$ to denote the release of lock o at Line i of thread t. Column 2 represents the control-flow

TABLE 1: The execution trace for Fig. 1 under the test input $t1$ = $\{$x=0, y=-2, z=0$\}$ to cover the target $<$(5, $w(y)$), (16, $r(y)$)$>$.

Concrete trace	$R^{1.1}y \rightarrow L^{1.2}lk \rightarrow R^{1.3}x \rightarrow U^{1.9}lk \rightarrow R^{1.13}z \rightarrow L^{2.15} \rightarrow R^{2.16}y \rightarrow W^{2.16}x \rightarrow U^{2.17}lk \rightarrow L^{2.18}lk \rightarrow R^{2.19}x \rightarrow R^{2.19}y \rightarrow U^{2.21}lk$
Symbolic trace	$R^{1.1}y < 0 \rightarrow L^{1.2}lk \rightarrow R^{1.3}x \leq 3 \rightarrow U^{1.9}lk \rightarrow R^{1.13}z < 3 \rightarrow L^{2.15} \rightarrow W^{2.16}x = R^{2.16}y$ - 1 $\rightarrow U^{2.17}lk \rightarrow L^{2.18}lk \rightarrow R^{2.19}x < 3 \rightarrow R^{2.19}y \neq 1 \rightarrow U^{2.21}lk$
$\phi_{pc_{W^{1.5}y}}$	$R^{1.1}y < 0 \wedge R^{1.3}x \leq 3$ ~~$\wedge R^{1.13}z < 3$~~ $\overset{\text{negate}}{\Longrightarrow} R^{1.1}y < 0 \wedge R^{1.3}x > 3$
$\phi_{pc_{R^{2.16}y}}$	~~$R^{2.19}x < 3 \wedge \rightarrow R^{2.19}y \neq 1$~~ // (16, $r(y)$) is covered
ϕ_{sync}	$((U^{1.4}lk \prec L^{2.15}lk) \wedge (U^{1.4}lk \prec L^{2.18}lk)\ ) \vee ((U^{2.17}lk \prec L^{1.2}lk) \wedge (U^{2.21}lk \prec L^{1.2}lk)\ ) \vee ((U^{2.17}lk \prec L^{1.2}lk) \wedge (U^{1.4}lk \prec L^{2.18}lk)\ )$
ϕ_{rw}	$((R^{1.1}y = y \wedge (R^{1.1}y \prec W^{2.20}y)) \vee (R^{1.1}y = W^{2.20}y \wedge (W^{2.20}y \prec R^{1.1}y))) \wedge ((R^{1.3}x = x \wedge (R^{1.3}x \prec W^{2.16}x)) \vee (R^{1.3}x = W^{2.16}x \wedge (W^{2.16}x \prec R^{1.3}x))) \wedge ((R^{1.5}x = x \wedge (R^{1.5}x \prec W^{2.16}x)) \vee (R^{1.5}x = W^{2.16}x \wedge (W^{2.16}x \prec R^{1.5}x)))$
Solution to $\Phi = \phi_{pc_{W^{1.5}y}} \wedge \phi_{pc_{R^{2.16}y}} \wedge \phi_{sync} \wedge \phi_{rw}$	$t_{new} = \{x = 5,\ y = -5,\ z = 0\}$; $s_{new} = R^{1.1}y \rightarrow L^{1.2}lk \rightarrow R^{1.3}x \rightarrow U^{1.4}lk \rightarrow R^{1.5}x \rightarrow W^{1.5}y \rightarrow L^{2.15} \rightarrow R^{2.16}y \rightarrow W^{2.16}x \rightarrow U^{2.17}lk \rightarrow L^{2.18}lk \rightarrow U^{2.21}lk$

decision: $R^{1.3}x \leq 3$ means the value of x read by Thr_1 at Line 3 is less than or equal to 3.

4.2.2 Constraint Modeling

The ordering constraint Φ encodes thread interactions, whose solution consists of a set of inputs and a schedule represented by a sequence of concurrent events. We allow two types of variables inside Φ: symbolic values returned by shared-variable reads, and ordering relation (i.e., $\prec$) that captures the ordering of global events. Thus, Φ is the conjunction of three constraints:

$$\Phi = \phi_{path} \wedge \phi_{sync} \wedge \phi_{rw},$$

where ϕ_{path} encodes the path condition constraints for all threads, ϕ_{sync} encodes the ordering constraints determined by synchronizations, and ϕ_{rw} encodes shared-memory constraints. Although it is possible to express the various memory consistency models [3], [13], [25], [34] in a similar fashion, in this work, we focus on the *sequential-consistency* memory only. That is, we assume statements within each thread are executed in the same order as they appear in the program.

Path Constraints (ϕ_{path}). *ConTesa* collects path constraints during the symbolic execution of individual threads. For each thread, it records the outcome of every branching predicate encountered during the execution. At the end of the execution, it tries to negate the branching predicate to cover an event in the target DU pair. If a certain event has already been covered, *ConTesa* removes the corresponding path predicate from consideration. Overall, ϕ_{path} denotes the conjunction of path constraints computed across all threads.

For example, in Table 1, the third row shows the path condition for the thread to reach the branch of $W^{1.5}y$ in the target DU pair. Note that *ConTesa* negates the condition of this particular branch; it also removes all the subsequent path conditions since they may no longer be needed. The fourth row shows the path condition for the thread to reach the read event $R^{2.16}y$ in the target DU pair. Since this event has been covered, all the subsequent path conditions are also removed.

Synchronization Constraints (ϕ_{sync}). *ConTesa* collects two types of synchronization constraints: partial order constraints and locking constraints. The partial order constraints capture the must-happen-before ordering of operations from concurrently running threads, such as thread *fork/join* and *wait/signal* operations. For example, all events executed after the *wait* must happen after all events before the corresponding *signal* operation. In contrast, the locking constraints coming from *lock* and *unlock* operations are modeled differently. Let the lock and unlock operations in thread Thr_1 be represented by L and U, respectively, and the lock and unlock operations in thread Thr_2 be represented by L' and U', respectively. Then, the locking constraints involve the following two possible cases. In the first case, thread Thr_1 acquires L first, so U happens before L'. In the second case, Thr_2 acquires the lock first, so U' happens before L. Thus, for each pair of lock-unlock guarded critical sections, the disjunctive formula that becomes part of ϕ_{sync} is composed of two constraints, representing the alternation of the above two cases. In Table 1, specifically, the fifth row shows the locking constraints for our running example, which has two lock/unlock pairs in the two threads. In this table, we use $\prec$ to denote the must-happen-before ordering relation between two events.

Read-Write Constraints (ϕ_{rw}). *ConTesa* also collects the read-write constraints to model all possible thread interactions through the shared memory. In general, a read from the shared memory may get the value written by a write in the same thread, or values written by different threads, depending on the order of the read and the writes. Thus, ϕ_{rw} is constructed as follows: for every read operation r on a variable v, if r is matched to a write w of the same variable, w must happen before r and there is no other write between them. In Table 1, for instance, the sixth row shows a read-write constraint for our running example: if $R^{1.1}y$ reads directly from the input (Line 1), the write must happen after the read operation ($R^{1.1}y \prec W^{2.20}y$).

In *ConTesa*, we invoke an off-the-shelf SMT solver to compute a solution for each symbolic variable that maps every `read` to a certain `write` on the shared memory under partial order constraints ϕ_{sync} and ϕ_{rw}. The last column of Table 1 shows the results. The size of our symbolic constraint, in the worst case, is linear in the number of conditional branches and cubic in the number of shared variable accesses in the execution trace.

4.3 Implementation

We have implemented *ConTesa* in a software tool that builds upon a number of open-source platforms. Specifically, we implemented `ImpAnalyzer` based on Diffutils and CodeSurfer [20]. We implemented `InvSelector` in C++. The `SchedExplorer` component was built upon the PIN [37] dynamic instrumentation and analysis framework.

The `TestAugmentor` component was built upon both PIN and KLEE [6], a symbolic virtual machine for C/C++. Specifically, we used KLEE to perform symbolic execution based test input generation and PIN to enforce the thread interleaving schedules. We also implemented the trace collection component as an LLVM function pass, which, similar to the Java front-end of CLAP [25], records a basic-block trace per thread and then uses the symbolic execution engine in KLEE to generate the entire trace. Although our implementation was based on LLVM, other similar technique for collecting concrete path profiles and generating symbolic traces may be used as well.

5 Empirical Study

We aim to answer the following research questions:

RQ1: How effective is *ConTesa* in augmenting test cases to complete the interleaving coverage and detect concurrency bugs in the new program P'?

RQ2: How efficient is *ConTesa* in generating new test cases compared to state-of-the-art test generation tools for concurrent software?

5.1 Objects of Analysis

We conducted experiments on thirteen multithreaded C/C++ applications. Three of the benchmarks, `bbuf`, `swarm` and `canneal`, are from [27], where `bbuf` is a reference implementation of a shared buffer, `swarm` implements a parallel sort, and `canneal` is a parallel implementation of the simulated annealing algorithm to minimize the routing cost of a chip design. Among the remaining benchmarks, `pfscan` is a multi-threaded file scanner, which combines the functionality of `find`, `xargs`, and `fgrep`; `aget` is a download accelerator that spawns multiple threads to download different chunks of a file in parallel; `pbzip2` is a parallel implementation of `bzip2`, which does file compression and file decompression; `transmission` is a BitTorrent client software; `cherokee` is a HTTP server; `memcached` is a general-purpose in-memory distributed caching system, which is designed to speed up websites by caching commonly requested data to ease back-end processing and database loads; `apache` is a web server that accepts configuration files on both client and server sites, and multiple main entry points with command line options. The last three benchmarks are open-source implementations of lock-free data structures [40].

For each of the benchmark applications, we utilized two program versions. In each case, the code changes leading to the modified version contain a real concurrency bug. By concurrency bug, we mean the program failure is caused exclusively by incorrectly protected thread interactions as opposed to errors in the sequential computation. For the first four and the last three benchmark applications, since there were no multiple versions available online, we used the downloaded (and buggy) version as the new program, and a fixed version as the original program.

Table 2 shows the statistics of these benchmark applications, containing the name of each application, the two program versions, the bug sources and types, the number of lines of non-comment code (NLOC), and the number of threads. The bugs in the 13 applications involve four deadlocks, eight order violations, and one atomicity violations. This also suggests that order violations are more common than the other types of bugs in these benchmark programs.

To answer the research questions in a *statistically significant manner*, we also need a set of existing inputs in the test suite. We wish to evaluate the effectiveness and efficiency of our technique when using test suites of different sizes. Toward this end, we create N test suites for each benchmark application, where a test suite consists of M test inputs. Column 6 of Table 2 lists the number of test inputs (M) together with the number of test suites (N), denoted by the pair (M, N), indicating there is a total number of $M \times N$ test inputs. We created more test suites and test inputs for the second set of benchmark programs because they are larger and more complex compared to the other benchmark programs. We inserted an assertion statement to the faulty location of each benchmark program.

Since the benchmark programs are not shipped with system tests that can test the functionality of the program, we need to generate tests for them. Specifically, the test inputs for `bbuf` are the number of consumers and producers; these inputs are randomly generated. For `swarm`, the test inputs are randomly generated arrays. For `canneal`, the test inputs are randomly generated integers. For `pfscan`, the test inputs are strings and files that we created to search a random string from a randomly chosen file or directory in each test run. For `aget`, the test inputs are randomly chosen files to be downloaded from the Internet specified by a predefined set of URLs. For `pbzip2`, the test inputs are some random files that we compressed *a priori* with different combinations of options. For `memcached`, the test inputs are chosen from some manually written test cases for performing operations such as set/get keys and incr/decr keys. For `transmission`, the test inputs are randomly downloaded torrents fed into the program under different configurations. For `cherokee` and `apache`, the test inputs are commands for issuing a session of requests to a set of static web pages using `httperf` under a given configuration. For the three `nbds` programs, the test inputs are randomly generated data fed to different data structures.

Statistics in the last three columns of Table 2 will be presented later in this section.

5.2 Variables and Measures

5.2.1 *Independent Variables*

Our independent variable involves are techniques used in the study. We wish to determine if *ConTesa* is cost-effective, and ideally such an assessment involves comparisons with state-of-the-art tools. However, since there is no prior work on regression test augmentation (RTA) for concurrent software, we instead compare to three somewhat related techniques.

The first technique is *Conc-iSE* [21], which employs an incremental symbolic execution algorithm to generate test inputs and interleaving schedules. However, *Conc-iSE* does not reuse test suites for guiding symbolic execution or schedule exploration, but instead generates all the test inputs from scratch.

TABLE 2: **Characteristics of objects and runtime statistics**

Name	Version	Bug	NLOC	# Threads	Tests	% Impact	% DUP$_{\texttt{aff}}$	% Cov DUP$_{\texttt{aff}}$
bbuf	v1	[27]	255					
	v2	deadlock	257	2	(50, 10)	3	9.1	72.2
swarm	v1	[27]	1636					
	v2	order	1638	5	(50, 10)	4	3.3	65.3
canneal	v1	[27]	2822					
	v2	deadlock	2833	2	(50, 10)	5	3.6	80.9
pfscan	v1	[44]	960					
	v2	deadlock	962	3	(50, 10)	5	2.5	80.2
aget [1]	0.4.1	[67]	850					
	0.4	atomicity	858	3	(50, 10)	2	3.4	77.5
pbzip2 [5]	1.1.2b3	[67]	3712					
	1.1.5	order	4069	4	(50, 10)	11	7.9	79.2
cherokee [10]	0.4.0	[67]	20204					
	0.4.1	order	20430	6	(250, 50)	19	11.4	30.2
memcache	1.4.3	[67]	26550					
	1.4.4	order	26599	7	(250, 50)	8	8.8	29.5
transmission	1.41	[67]	154264					
	1.42	order	154393	6	(250, 50)	14	12.4	25.4
apache	2.0.47	[67]	95952					
	2.0.48	order	97153	6	(250, 50)	21	14.5	31.2
nd-ls [40]	v1	[21]	1629					
	v2	order	1770	5	(50, 10)	6	5.4	72.4
nd-sl [40]	v1	[21]	2091					
	v2	order	2112	5	(50, 10)	4	3.2	78.5
nd-ht [40]	v1	[21]	2234					
	v2	order	2325	5	(50, 10)	5	6.1	71.2

The second technique is *Con2colic* [17], a tool for generating test inputs and thread interleavings for concurrent software. Since the tool is not publicly available, we re-implemented it in our own framework. A difference between *Con2colic* and *ConTesa* is *Conc2colic* does not reuse existing test suites.

The third technique is *ReConTest* [57], which selects new interleavings of existing test inputs that contain at least one of the affected accesses. However, it does not generate new test inputs. Note that deactivating symbolic execution of *ConTesa* does not get *ReConTest* because *ConTesa* uses symbolic execution to generate both new inputs and new interleavings, whereas *ReConTest* uses existing test inputs to explore new interleavings. Since the original *ReConTest* tool only handles Java programs, we also re-implemented it to handle C/C++ applications.

5.2.2 Dependent Variables

Our dependent variables are the metrics chosen to measure the effectiveness and efficiency of *ConTesa* and the other techniques. In terms of effectiveness, we measure, for the inter-thread DU pairs affected by code changes, the percentage covered by tests resulting from each of the aforementioned tools. To account for possible differences in coverage per execution of the initial test suites, we executed each object on the various sets of initial test suites, and calculated the average. We then compared the number of concurrency failures detected by each tool.

In terms of efficiency, we first measured the average number of inputs generated by different techniques. We then measured the total *testing time* of each tool, together with a breakdown in terms of the time required for impact analysis (if any), for generating inputs (if any) and interleaving schedules, and for replaying the program. Finally, we measured the time it took to detect the fault for each technique. To account for possible differences in the testing times per execution of initial test suites, we executed each object on the various initial test suites and then calculated the average.

5.3 Study Operation

Column 7 shows the number of DU pairs impacted by the code changes in modified program versions. Column 8 shows the percentage of the affected DU pairs over all DU pairs in the program. Column 9 reports the affected DU pairs covered by the existing test cases. During our experiments, we set a time-budget of 12 hours for each tool, which is in line with the use of regression testing in practice (e.g., nightly-build-and-test). The maximum time for the solver (i.e., the option `--max-solver-time`) in KLEE is set to 300 seconds. For each pair of tool comparisons, we applied a t-test to the coverage (cost) data and used 0.05 as the confidence level to determine whether there is a statistically significant difference between two techniques.

6 Results

Table 3 and Table 4 show the results, including the testing coverage and the number of faults detected by each technique.

6.1 RQ1: Effectiveness of *ConTesa*

RQ1 involves the effectiveness of *ConTesa* in obtaining high coverage of the *affected* DU pairs and the related faults.

Coverage. The average test coverage, as measured by the affected DU pairs, ranges from 71.1% to 100% as shown in Columns 2-5 of Table 3. For eight benchmark programs, *ConTesa* achieved 100% coverage. On nine out of the thirteen programs, *Conc2colic* achieved the same coverage as *ConTesa*, but on four larger applications, namely `cherokee`, `memcache`, `transmission` and `apache`, *ConTesa* achieved significantly higher coverage. When comparing *ConTesa* to

TABLE 3: Effectiveness of *ConTesa* compared to three state-of-the-art testing techniques.

Name	*Testing Coverage (%)*				*Faults Detected*			
	ConTesa	Conc-iSE	Con2colic	ReConTest	*ConTesa*	Conc-iSE	Con2colic	ReConTest
bbuf	100	100	100	100	✓	✓	✓	✓
swarm	100	100	100	100	✓	✓	✓	✓
canneal	100	100	100	0	✓	✓	✓	✗
pfscan	100	100	100	100	✓	✓	✓	✓
aget	100	100	100	0	✓	✓	✓	✗
pbzip2	89.2	89.2	89.2	58.0	✓	✓	✓	✓
cherokee	82.3	70.8	41.4	64.9	✓	✓	✗	✗
memcache	88.5	69.4	41.3	68.6	✓	✓	✗	✗
transmission	76.5	67.5	45.4	54.7	✓	✗	✗	✓
apache	71.1	58.8	34.0	41.3	✓	✗	✗	✗
nbds-list	100	100	100	63.2	✓	✓	✓	✓
nbds-slist	100	100	100	70.0	✓	✓	✓	✗
nbds-htable	100	100	100	71.5	✓	✓	✓	✗

TABLE 4: Efficiency of *ConTesa* compared to three state-of-the-art testing techniques.

Name	*# Test Inputs*				*Executing Time (seconds)*				*Detection Time (seconds)*			
	ConTesa	Conc-iSE	Con2colic	ReConTest	*ConTesa*	Conc-iSE	Con2colic	ReConTest	*ConTesa*	Conc-iSE	Con2colic	ReConTest
bbuf	0	2	6	0	2.1	3.4	19.5	1.23	1.8	2.5	10.6	1.6
swarm	2	5	8	0	19.5	50.8	1159.2	13.8	11.4	29.4	395.6	10.5
canneal	0	2	5	0	39.4	74.5	99.2	18.3	7.9	32.2	85.5	-
pfscan	0	3	7	0	19.4	91.2	201.1	12.3	12.3	49.2	98.9	10.9
aget	3	4	9	0	209.1	392.4	462.4	121.5	66.3	70.2	101.5	-
pbzip2	3	7	15	0	50.5	80.2	166.2	33.2	28.5	39.5	91.3	19.6
cherokee	6	21	49	0	3059.7	5948.5	>	1198.0	1211.5	2648.8	-	-
memcache	21	56	88	0	1542.5	3021.4	>	316.5	682.4	988.9	-	-
transmission	19	33	139	0	8482.9	>	>	1098.1	2138.4	-	-	477.2
apache	16	42	156	0	6704.6	>	>	2125.0	4985.6	-	-	-
nbds-list	2	9	19	0	421.2	692.8	2022.1	102.4	308.4	392.5	1148.7	42.3
nbds-slist	3	12	25	0	322.3	588.6	1674.4	158.2	262.0	407.6	1692.4	-
nbds-htable	3	11	27	0	205.5	386.7	1842.1	144.2	178.3	308.5	1596.6	-

ReConTest, we found that on ten out of the thirteen programs, *ConTesa* was significantly better, whereas on three programs, they were equally effective – this was because all the affected DU pairs in these three programs could be exercised by existing test inputs. When comparing *ConTesa* to Conc-iSE, we found that *ConTesa* performed better on four programs: `cherokee`, `memcache`, `transmission`, and `apache`, indicating the use of existing test suites can significantly increase the performance, particularly on large software.

Fault detection. The numbers of faults detected by *ConTesa* and the three competing techniques are shown in Columns 6-9 of Table 3. Specifically, *ConTesa* detected all of the thirteen concurrency bugs. For the programs where *ConTesa* did not achieve 100% coverage, the reason is due to known limitations of KLEE: for example, in some cases, covering specific DU pairs requires the program inputs to be hierarchical file directories, but KLEE models the file system as a flattened system, where symbolic files can only have pathnames such as “A”, “B”, and “C” without any hierarchy. Other cases were due to timeouts KLEE encountered when solving the hard formulas.

In contrast, *Conc-iSE* detected eleven, *Con2colic* detected nine, and *ReConTest* detected six. These results indicate that *ConTesa* is more effective in detecting concurrency faults than state-of-the-art techniques.

6.2 RQ2: Efficiency of *ConTesa*

RQ2 involves the number of generated test inputs and time taken by *ConTesa* to obtain high coverage of the *affected* DU pairs.

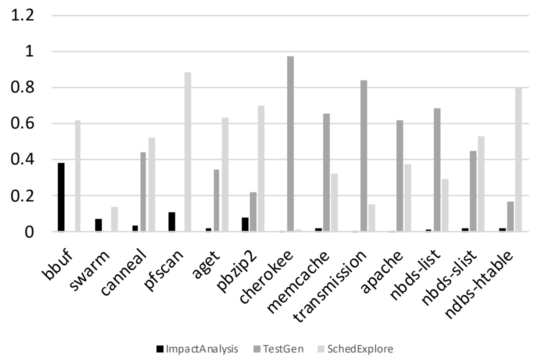

Fig. 3: Percentage of the running time spent on impact analysis, schedule exploration, and test case generation.

Columns 2-5 of Table 4 show the the number of test inputs generated for the four techniques. Overall, *ConTesa* generates 61.6% less test inputs than *Conc2colic* and 85.9% less test inputs than *Con2Colic*. Since *ReConTest* does not generate new inputs, so the numbers are all 0s.

Columns 6-9 show the total execution time of the four techniques on achieving the coverage of affected DU pairs. Overall, *ReConTest* required the least amount of time because it does not need to augment test cases. *ConTesa* was the second fastest and completed on all thirteen benchmark applications. In contrast, *Conc-iSE* timed out on two applications and *Conc2colic* timed out on four applications. Furthermore, on the nine program for which all techniques completed, *ConTesa* was 2.5x to 39.3x faster than *Con2Colic*, and 1.5x to 1.9x faster than *Conc-iSE*. These results indicate that *ConTesa* is more efficient. When comparing *ConTesa* to

ReConTest, we found that on the six programs in which *ConTesa* did not call `TestAugmentor`, *ConTesa* required more time because it needed to generate the symbolic traces, which introduced extra overhead. On the other five programs, *ConTesa* required more time because it invoked `TestAugmentor` to generate inputs. But overall, ReConTest achieved 34.4% less coverage and detected 53.8% less concurrency bugs. Considering the effectiveness of *ConTesa*, its additional cost is acceptable.

Figure 3 displays the results regarding the percentage of time used in running impact analysis, schedule exploration, and test input generation. Overall, the time for impact analysis never exceeded 33 seconds, which accounted for only 0.67% of technique runtime overall. The times for schedule exploration and test case input are 23.8% and 75.5% of the overall runtime, respectively.

The last four columns of Table 4 show the time spent on detecting the faults for each technique. The symbol "-" indicates no faults were detected. On average, on the 11 programs that both *ConTesa* and *Con2Colic* detected the faults, *ConTesa* was 1.8x faster than *Con2Colic* to expose the faults. Compared *ConTesa* to *Con2Colic*, *ConTesa* was 6x faster. On the six programs in which faults were detected by both *ConTesa* and *ReConTest*, *ConTesa* was 6x slower than *ReConTest*. Again, these results suggest that *ConTesa* is more cost-effective than the other three techniques.

6.3 Threats to Validity

The primary threat to *external validity* is the representativeness of our objects and test cases, since other objects and test cases may exhibit different behaviors and cost-benefit tradeoffs. However, the threat has been significantly reduced by our use of reputable open-source objects from a variety of previously published studies, as well as a large number of initial test suites. The primary threat to *internal validity* is possible defects in the implementation of our own tool as well as the tools we re-implemented to perform the experimental evaluation on C/C++ applications. We have been careful in our implementation, used extensive testing as well as manual inspection to determine the correctness of the experimental results. In terms of the *construct validity*, there are other metrics that could be pertinent to the effects studied. In particular, our measurements of cost consider only the execution time of the tool while omitting the time taken by engineers to use the tool. Our time measurements also suffer from the potential biases discussed under *internal validity*, given the inherent difficulty in obtaining an efficient tool prototype.

7 Summary and Discussions

7.1 Summary of Results

ConTesa was more effective (in terms of coverage and fault detection) and efficient (in terms of execution time and fault detection time) than Conc-ise, Con2colic. Compared to Con2colic and Conc-ise, the main advantage of *ConTesa* is the reuse of existing test inputs for guiding symbolic execution, which scales better on large programs. *ConTesa* was more effective than ReConTest, because *ConTesa* augments existing test inputs to cover more program elements related to concurrency faults. While *ConTesa* was less efficient than ReConTest due to the additional time for generating inputs, the cost is acceptable considering the benefits of high coverage and fault detection rate.

TABLE 5: Impact of implementation choices inside *ConTesa*.

Name	*BFS Search vs Random*		*Existing vs New Method*	
	Coverage ↑	Time ↓	Coverage ↑	Time ↓
Bbuf	=	R (4.2%)	=	E (3.5%)
Swarm	=	B (6.4%)	=	E (7.8%)
Canneal	=	B (6.9%)	=	E⋆ (5.2%)
Pfscan	=	B (6.6%)	=	E⋆ (9.3%)
Aget	=	B (4.5%)	=	E⋆ (6.8%)
Pbzip2	=	B (8.9%)	=	E⋆ (4.2%)
Cherokee	=	B⋆ (5.5%)	N⋆ (11.3%)	E⋆ (3.2%)
Memcache	=	B⋆ (6.9%)	N⋆ (3.2%)	E⋆ (4.2%)
Transmission	=	B⋆ (3.6%)	N⋆ (5.8%)	E⋆ (9.7%)
Apache	=	B⋆ (2.2%)	=	E⋆ (8.2%)
nbds-list	=	B (1.2%)	=	E (2.8%)
nbds-skiplist	=	B⋆ (3.3%)	=	E⋆ (2.5%)
nbds-hashtable	=	B⋆ (2.1%)	=	E⋆ (1.9%)

7.2 Discussions

We now explore additional observations relevant to our study.

Ordering affected program entities. We also investigated, within *ConTesa*, the impact of exploring the program entities in different orders. Specifically, we compared the performance of using the default breadth-first search (BFS) order with random order. Results of this comparison are shown in Columns 2 and 3 of Table 5, where the testing coverage is shown in Column 2 and the testing time is shown in Column 3. Each entry summarizes the differences observed: "B" means *BFS* achieved a greater mean coverage or faster time, "R" means *Random* achieved a greater mean coverage or faster time, and "=" means the two exhibited equal mean coverage. The symbols marked as ⋆ mean the differences were statistically significant. The numbers in the parentheses indicate the improvements of coverage and the reductions of the runtime cost.

These results show that, in terms of the testing coverage, the order of exploration does not significantly affect the effectiveness of the algorithm. This is reasonable because the same program elements would ultimately be considered under any order. However, in terms of the testing time, the order of exploration does make significant differences. Our results show that *BFS* often provides savings in the execution time. This is because test cases that cover branches higher in dependency chains would have inputs close to those used to reach lower branches, thus seeding their inputs to help *ConTesa* reach the targets more quickly. Overall, *BFS* seems to be more efficient, but these results do not preclude other orderings that may be even more cost-effective.

Existing and new test inputs. We considered two cases of reusing test inputs in *ConTesa*: 1) reusing only the existing test inputs, and 2) reusing both existing and newly-generated test inputs. The experimental comparison of these two approaches was shown in Columns 4-5 of Table 5, where "E" means the first option achieved better results and "N" means the second option achieved better results. In terms of testing coverage, the second option is slightly better, whereas in terms of time, the first option is consistently better.

The results show that reusing newly generated test inputs increases the testing cost, with mostly marginal benefit. It highlights the tradeoff between achieving higher coverage and reducing the testing time. Our conjecture is that, on larger programs, the gain in testing effectiveness may be worth the slowdown in the testing time.

Effectiveness of existing test inputs. We next discuss how different existing test suites can affect the effectiveness and efficiency of *ConTesa*. We consider four cases: 1) the set of existing test inputs is empty, 2) the set of existing test inputs covers all affected DU pairs, 3) the set of existing test inputs is not empty, but it does not contain any failure-inducing inputs, and 4) the set of existing test inputs is not empty and it contains one failure-inducing input and it does not cover all affected DU pairs.

In the first case, *ConTesa* and *Conc-iSE* behave equivalently because *ConTesa* needs to generate all new test inputs and interleaving schedules to cover affected DU pairs. In the second case (an idea case), *ConTesa* is equal to *ReConTest* because both techniques only need to exercise existing test inputs without generating new inputs or new interleavings for achieving the coverage.

In the third case, *ReConTest* would not be able to detect faults because new test inputs are needed to exercise the faulty DU pairs. To compare *ConTesa* with *Conc-iSE* and *Con2Colic*, we removed all failure-inducing inputs from the test suites generated in our empirical study. The results showed that *ConTesa* was still 1.6x faster than *Conc-iSE* and 5.8x faster than *Con2Colic* on exposing the faults. This is because *Conc-iSE* and *Con2Colic* required generating new inputs to to cover the affected DU pairs before reaching the faulty pair.

In the fourth case, we manually created a failure-inducing input for each subject and added it to the test suites. The results showed that *ConTesa* was much more faster than *Conc-iSE* (2.3x) and *Con2Colic* (6.8x). Because of the failure-inducing inputs, both *ConTesa* and *ReConTest* detected equal number of faults and *ReConTest* was 6.2x faster than *ConTesa*. However, *ConTesa* achieved 47.8% more coverage than *ReConTest*.

Application of *ConTesa*. By design, *ConTesa* is cost-effective only when the code changes between P and P' affects a small subset of the entire program. If the entire program is affected, the incremental analysis in *ConTesa* will degenerate to the non-incremental one with little advantage over traditional techniques. Therefore, *ConTesa* is the most suitable for development environments where the correctness of frequent but small code changes is checked before committed to the repository. In our experiments, 10 out of the 13 applications are developer-made code modifications that meet the aforementioned criterion: these code modifications affected 0.3% to 10.3% of the program entities. Therefore, they reflect at least some real-world software development scenarios in practice. However, it remains an open question whether they reflect the vast majority of real-world software development scenarios.

Interleaving coverage criteria. Interleaving coverage criteria may impact how well *ConTesa* works. Lu et. all [36] introduced seven interleaving coverage criteria, which are designed based on different concurrency fault models. Their cost ranges from exponential to linear. Study by Hong et al. [23] also confirmed the effectiveness of concurrency fault detection can vary depending on such criteria. For the three baseline tools compared in our work, *Conc-iSE* does not measure coverage, *Con2colic* measures branch coverage, and *ReConTest* measures access patterns violating atomic-set serializability. While *ConTesa* employs Def-Use criteria by default, as part of the future work, it is important to investigate the cost-effectiveness of RTA under other criteria [63] in the context of regression testing.

8 Related Work

There is a large and growing body of work on testing concurrent programs [4], [11], [18], [29], [38], [39], [41], [46], [52], [53], [61], [62], [64], but most existing techniques do not consider the regression testing of evolving software.

Among the few techniques that target regression testing for concurrent programs, none of them solves the regression test augmentation (RTA) problem. SimRT [69] is a test case selection and prioritization framework for concurrent programs. However, it does not generate new test inputs or reduce the interleaving exploration cost inherent in testing. ReConTest [57] addresses this problem by selecting new interleavings that arise due to code changes but may miss accesses not exercised by existing test inputs. Conc-iSE [21] is an incremental symbolic execution algorithm for concurrent software, which leverages execution summaries [22] to prune away previous explored execution traces. While it is capable of generating new test inputs and thread schedules, it does not reuse existing test cases to explore the affected interleaving space or guide the input generation.

In a position paper we published earlier [68], we envisioned a general framework within which regression test augmentation (RTA) may be implemented for multithreaded programs, to reuse existing test suites as well as generate new test cases. However, the framework was not yet realized or evaluated. In this work, we developed the initial idea, implemented what is believed to be the first RTA tool for multithreaded C/C++ programs, and evaluated it on a large set of real-world applications.

There are change-impact analysis techniques designed with a particular focus on multi-threaded programs [9], [26], [58], [69]. There are also change-impact analysis techniques for distributed systems [2], [7], [45], [59], [60]. Although these techniques may result in different costs and benefits, and in principle, may be leveraged by *ConTesa*, they do not address the RTA problems themselves.

There are also testing techniques [12], [38], [63], such as CHESS, that select and prioritize the interleaving schedules in concurrent software to expose bugs more quickly. Other techniques have been geared toward systematically exploring the thread schedules in multi-threaded programs across program versions [19], [26]. In particular, Gligoric et al. [19] reuse results from the exploration of one program version to speed up the exploration of the next program version. Jagannath et al. [26] use information about program changes in software evolution to prioritize the exploration of schedules. These techniques, however, target exploration of schedules within individual test cases and do not address the challenges of regression testing involving large sets of test cases.

More recently, Deng et al. [14] experimentally studied how well various existing concurrency fault detection tools perform for a set of test inputs. They also proposed a technique that first measures coverage of a program, and then selects a subset of test inputs to test for data races and atomicity violations on that program. However, their technique focuses on a single program version and does not consider code changes. Also, like SimRT [69] and ReConTest [57], their technique relies on existing test inputs. In contrast, the main contribution of our work is to leverage the code changes to more effectively reuse existing test suites as well as generate new test cases.

At the same time, there are many techniques for test suite augmentation [8], [43], [50], [56], [65]. Santelices et al. [50] combines dependence analysis and symbolic execution to identify chains of data and control dependencies that, if tested, are likely to exercise the effects of changes. Person et al. [43] presents a differential technique that uses symbolic execution to identify affected elements more precisely than [50], and yields constraints that can be input to a solver to generate test cases for those requirements. However, these techniques focus only on sequential programs; they are incapable of effectively identifying affected concurrency program elements or generating inputs or thread interleavings to exercise concurrency-related new program behaviors.

9 Conclusions and Future Work

We have presented *ConTesa*, a regression test augmentation tool for concurrent software, capable of reusing the existing test suites as well as generating new test cases. It treats the test input generation and interleaving exploration problems uniformly, in which new test inputs are generated from test reuse to guide the exploration of affected interleaving space not yet covered by existing inputs. It can also replay regression concurrency faults by leveraging an active scheduler. We have evaluated *ConTesa* on a set of multithreaded Linux applications. Our results show that it outperforms state-of-the-art techniques in terms of the execution time, testing coverage, and fault-detection capability.

There are other test case generation techniques that could be used to address the RTA problem. In this work, we choose to focus on a dynamic technique that leverages existing test cases. Other dynamic techniques, such as evolutionary or search-based approaches, are also known to be effective in addressing RTA for sequential problems [65]; as part of our future work, we will evaluate these dynamic test case generation techniques. In addition, we plan to perform more extensive experiments on additional sets of benchmark programs.

References

[1] AGET. Multithreaded HTTP Download Accelerator. Web page. http://www.enderunix.org/aget/.

[2] K. A. Alam, R. Ahmad, A. Akhunzada, M. H. N. M. Nasir, and S. U. Khan. Impact analysis and change propagation in service-oriented enterprises: A systematic review. *Information Systems*, 54:43–73, 2015.

[3] J. Alglave, D. Kroening, and M. Tautschnig. Partial orders for efficient bounded model checking of concurrent software. In *International Conference on Computer Aided Verification*, pages 141–157, 2013.

[4] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 167–178, 2010.

[5] BZIP2. Parallel BZIP2 . Web page. http://compression.ca/pbzip2/.

[6] C. Cadar, D. Dunbar, D. R. Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation*, volume 8, pages 209–224, 2008.

[7] H. Cai and D. Thain. DISTEA: efficient dynamic impact analysis for distributed systems. *Computing Research Repository*, abs/1604.04638, 2016.

[8] J. Campos, A. Arcuri, G. Fraser, and R. Abreu. Continuous test generation: enhancing continuous integration with automated test generation. In *Proceedings of the ACM/IEEE international conference on Automated software engineering*, pages 55–66, 2014.

[9] K. Chatterjee, L. De Alfaro, V. Raman, and C. Sánchez. Analyzing the impact of change in multi-threaded programs. In *International Conference on Fundamental Approaches to Software Engineering*, pages 293–307, 2010.

[10] Cherokee. Cherokee. Web page. http://cherokee-project.com.

[11] K. E. Coons, S. Burckhardt, and M. Musuvathi. GAMBIT: effective unit testing for concurrency libraries. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 15–24, 2010.

[12] K. E. Coons, S. Burckhardt, and M. Musuvathi. Gambit: Effective unit testing for concurrency libraries. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2010.

[13] B. Demsky and P. Lam. SATCheck: SAT-directed stateless model checking for SC and TSO. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 20–36, 2015.

[14] D. Deng, W. Zhang, and S. Lu. Efficient concurrency-bug detection across inputs. In *International Conference on Object-Oriented Programming, Systems, Languages and Applications*, 2013.

[15] H. Do and G. Rothermel. On the use of mutation faults in empirical assessments of test case prioritization techniques. *IEEE Transactions on Software Engineering*, 32(9):733–752, 2006.

[16] S. Elbaum, A. G. Malishevsky, and G. Rothermel. Test case prioritization: A family of empirical studies. *IEEE Transactions on Software Engineering*, 28(2):159–182, 2002.

[17] A. Farzan, A. Holzer, N. Razavi, and H. Veith. Con2Colic Testing. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 37–47, 2013.

[18] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 338–349, 2003.

[19] M. Gligoric, V. Jagannath, and D. Marinov. Mutmut: Efficient exploration for mutation testing of multithreaded code. In *Proceedings of the International Conference on Software Testing, Verification and Validation*, pages 55–64, 2010.

[20] GrammaTech. CodeSurfer. Web page. http://www.grammatech.com/products/codesurfer/overview.html.

[21] S. Guo, M. Kusano, and C. Wang. Conc-ise: Incremental symbolic execution of concurrent software. In *IEEE/ACM International Conference on Automated Software Engineering*, pages 531–542, 2016.

[22] S. Guo, M. Kusano, C. Wang, Z. Yang, and A. Gupta. Assertion guided symbolic execution of multithreaded programs. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 854–865, 2015.

[23] S. Hong, M. Staats, J. Ahn, M. Kim, and G. Rothermel. Are Concurrency Coverage Metrics Effective for Testing: A Comprehensive Empirical Investigation. *STVR*, 25(4):334–370, 2015.

[24] J. Huang and C. Zhang. Persuasive prediction of concurrency access anomalies. In *International Symposium on Software Testing and Analysis*, pages 144–154, 2011.

[25] J. Huang, C. Zhang, and J. Dolby. Clap: Recording local executions to reproduce concurrency failures. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 141–152, 2013.

[26] V. Jagannath, Q. Luo, and D. Marinov. Change-aware Preemption Prioritization. In *International Symposium on Software Testing and Analysis*, pages 133–143, 2011.

[27] N. Jalbert and K. Sen. A trace simplification technique for effective debugging of concurrent programs. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 57–66, 2010.

[28] J. A. Jones and M. J. Harrold. Test-suite reduction and prioritization for modified condition/decision coverage. *IEEE Transactions on Software Engineering*, 29(3):195–209, 2003.

[29] S. Joshi, S. K. Lahiri, and A. Lal. Underspecified harnesses and interleaved bugs. In *POPL*, pages 19–30, 2012.

[30] V. Kahlon, N. Sinha, E. Kruus, and Y. Zhang. Static data race detection for concurrent programs with asynchronous calls. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 13–22, 2009.

[31] V. Kahlon, Y. Yang, S. Sankaranarayanan, and A. Gupta. Fast and accurate static data-race detection for concurrent programs. In *International Conference on Computer Aided Verification*, pages 226–239, 2007.

[32] B. Korel, G. Koutsogiannakis, and L. H. Tahat. Application of system models in regression test suite prioritization. In *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*, pages 247–256, 2008.

[33] M. Kusano and C. Wang. Flow-sensitive composition of thread-modular abstract interpretation. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, 2016.

[34] M. Kusano and C. Wang. Thread-modular static analysis for relaxed memory models. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 337–348, 2017.

[35] Z. Li, M. Harman, and R. M. Hierons. Search algorithms for regression test case prioritization. *IEEE Transactions on Software Engineering*, 33(4):225–237, 2007.

[36] S. Lu, W. Jiang, and Y. Zhou. A Study of Interleaving Coverage Criteria. In *FSE companion*, pages 533–536, 2007.

[37] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 190–200, 2005.

[38] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing Heisenbugs in concurrent programs. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation*, pages 267–280, 2008.

[39] M. Naik, C.-S. Park, K. Sen, and D. Gay. Effective static deadlock detection. In *International Conference on Software Engineering*, pages 386–396, 2009.

[40] NBDS. Non-blocking data structures. Web page. https://code.google.com/p/nbds/.

[41] A. Nistor, Q. Luo, M. Pradel, T. R. Gross, and D. Marinov. Ballerina: Automatic generation and clustering of efficient random unit tests for multithreaded code. In *Proceedings of the International Conference on Software Engineering*, pages 727–737, 2012.

[42] A. Orso, N. Shi, and M. J. Harrold. Scaling regression testing to large software systems. In *Proceedings of the International Symposium on Foundations of Software Engineering*, volume 29, pages 241–251. ACM, 2004.

[43] S. Person, M. B. Dwyer, S. Elbaum, and C. S. Psreanu. Differential symbolic execution. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 226–237, 2008.

[44] Parallel file scanner. http://ostatic.com/pfscan, 1999.

[45] D. Popescu, J. Garcia, K. Bierhoff, and N. Medvidovic. Impact analysis for distributed event-based systems. In *Proceedings of the ACM International Conference on Distributed Event-Based Systems*, pages 241–251, 2012.

[46] M. Pradel and T. R. Gross. Fully automatic and precise detection of thread safety violations. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 521–530, 2012.

[47] G. Rothermel and M. J. Harrold. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, 22(8):529–551, 1996.

[48] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology*, 6(2):173–210, 1997.

[49] G. Rothermel, R. J. Untch, and C. Chu. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, 27(10):102–112, 2001.

[50] R. Santelices, P. K. Chittimalli, T. Apiwattanapong, A. Orso, and M. J. Harrold. Test-suite augmentation for evolving software. In *IEEE/ACM International Conference on Automated Software Engineering*, pages 218–227, 2008.

[51] R. Santelices, M. J. Harrold, and A. Orso. Precisely detecting runtime change interactions for evolving software. In *IEEE International Conference on Software Testing, Verification and Validation*, pages 429–438, 2010.

[52] K. Sen. Effective random testing of concurrent programs. In *Proceedings of International Conference on Automated Software Engineering*, pages 323–332, 2007.

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

[54] K. Sen and G. Agha. Cute and jcute: Concolic unit testing and explicit path model-checking tools. In *International Conference on Computer Aided Verification*, pages 419–423, 2006.

[55] A. Srivastava and J. Thiagarajan. Effectively prioritizing tests in development environment. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 97–106, 2002.

[56] K. Taneja, T. Xie, N. Tillmann, and J. de Halleux. express: guided path exploration for efficient regression test generation. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 1–11, 2011.

[57] V. Terragni, S.-C. Cheung, and C. Zhang. RECONTEST: Effective Regression Testing of Concurrent Programs. In *International Conference on Software Engineering*, 2015.

[58] V. Terragni, S.-C. Cheung, and C. Zhang. Recontest: Effective regression testing of concurrent programs. In *IEEE International Conference on Software Engineering*, volume 1, pages 246–256, 2015.

[59] S. Tragatschnig, H. Tran, and U. Zdun. Impact analysis for event-based systems using change patterns. In *Proceedings of the Annual ACM Symposium on Applied Computing*, pages 763–768, 2014.

[60] S. Tragatschnig and U. Zdun. Modeling change patterns for impact and conflict analysis in event-driven architectures. In *IEEE International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pages 44–46, 2015.

[61] W. Visser, K. Havelund, G. P. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, 2003.

[62] C. von Praun and T. R. Gross. Static conflict analysis for multi-threaded object-oriented programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 115–128, 2003.

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

[64] A. Williams, W. Thies, and M. D. Ernst. Static deadlock detection for java libraries. In *European Conference on Object-Oriented Programming*, pages 602–629, 2005.

[65] Z. Xu, Y. Kim, M. Kim, G. Rothermel, and M. B. Cohen. Directed test suite augmentation: techniques and tradeoffs. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 257–266, 2010.

[66] S. Yoo, M. Harman, P. Tonella, and A. Susi. Clustering test cases to achieve effective and scalable prioritisation incorporating expert knowledge. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 201–212, 2009.

[67] J. Yu, S. Narayanasamy, C. Pereira, and G. Pokam. Maple: A coverage-driven testing tool for multithreaded programs. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 485–502, 2012.

[68] T. Yu. Taco: Test suite augmentation for concurrent programs. In *Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 918–921, 2015.

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