TACO: Test Suite Augmentation for Concurrent Programs

Tingting Yu
Department of Computer Science
University of Kentucky
Lexington, Kentucky, 40506, USA
tyu@cs.uky.edu

ABSTRACT

The advent of multicore processors has greatly increased the prevalence of concurrent programs to achieve higher performance. As programs evolve, test suite augmentation techniques are used in regression testing to identify where new test cases are needed and then generate them. Prior work on test suite augmentation has focused on sequential software, but to date, no work has considered concurrent software systems for which regression testing is expensive due to large number of possible thread interleavings. In this paper, we present TACO, an automated test suite augmentation framework for concurrent programs in which our goal is not only to generate new inputs to exercise uncovered changed code but also to explore new thread interleavings induced by the changes. Our technique utilizes results from reuse of existing test inputs following random schedules, together with a predicative scheduling strategy and an incremental concolic testing algorithm to automatically generate new inputs that drive program through affected interleaving space so that it can effectively and efficiently validate changes that have not been exercised by existing test cases. Toward the end, we discuss several main challenges and opportunities of our approach.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools*

Keywords

Regression Testing, Concurrency

1. INTRODUCTION

Testing real world concurrent program is challenging primarily because concurrency faults are sensitive to execution interleavings. Unless a specific interleaving is exercised during testing that can cause faults to occur and cause their effects to be visible, they will remain undetected. Unfortunately, in real world concurrent programs, the interleaving space is often too large to be thoroughly explored [6]. To address this problem, numerous program analysis and testing techniques have been used, such as dynamic monitoring [2] and deterministic scheduling [7]. However, most of these techniques are concerned about single version of programs and do not consider evolved software.

Regression testing is used to perform re-validation of evolving software. To reduce cost of regression testing, various approaches have been suggested based on existing test cases, including regression test selection and test case prioritization. However, existing test cases may not be adequate to validate the code or system behaviors that are present in a new version of a system. Test suite augmentation (RTA) addresses this problem by identifying code elements affected by changes and generating test cases to cover those elements. For example, research on different flavors of static/dynamic symbolic execution has been shown to be effective at generating program inputs that execute modified code [10]. However, most research on test suit augmentation has focused on sequential programs, and to our knowledge none has considered the issues involved in augmenting test cases for concurrent programs.

Unlike sequential programs for which RTA considers only inputs, adapting RTA to concurrent programs requires generating both inputs and interleavings to explore changes introduced in the modified programs. In our prior work [11] we developed a regression testing framework focusing on selection and prioritization at the input level, by identifying affected shared variable (SV) accesses related to data races. Terragni et al. [8] further addresses regression testing of concurrent programs at interleaving level by searching only interleavings pertinent to affected SV accesses. The insight behind this approach is that only a small percentage of accesses (1% for real-world applications in their study) are affected; interleavings that do not contain affected accesses can be skipped in regression testing. While both SIMRT and RECONTEST reduce the cost of regression testing of concurrent programs, they focus on using existing test inputs and do not consider exploring affected interleaving space in which new inputs are needed.

In this paper, we present the first **T**est suite **A**ugmentation framework for **CO**ncurrent programs – TACO, that integrates approaches to test input augmentation with predicative scheduling techniques, focusing on program changes leading to potentially erroneous thread interleavings. TACO operates in three steps. First, TACO executes modified program by using existing test inputs under random schedules. For each trace obtained from a test execution, TACO identifies shared variable (SV) accesses affected by program changes. For a given affected SV pair covered in the trace, TACO actively seeks alternative thread interleavings related to this pair. The alternative interleavings are inferred by matching an interleaving coverage criterion. In the second step, TACO leverages concolic testing to generate new inputs that cover uncovered affected SV pairs; newly discovered pairs are explored in the same manner as Step 1.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

ESEC/FSE'15, August 30 – September 4, 2015, Bergamo, Italy
© 2015 ACM. 978-1-4503-3675-8/15/08...$15.00
http://dx.doi.org/10.1145/2786805.2803201

```
g1 = 1, g2 = 1;                    T2:
...                                ...
T1:                                8. mutex_lock(&lk);
...                                9. if (a>=0){
1.                                 10.    x = g1 + 1;
(&lk);                             11.    y = g2 + a; }
2. if ( b < 0)                     12. else
3.    g2 = 3;                      13.    x = a * g1;
4. else {                          14. cond_signal(&cond);
5.    cond_wait(&cond, &lk);--     15. mutex_unlock(&lk);
6.    g1 = 0; }                        ...
7. mutex_unlock(&lk);              16. if(b>3}
                                   17.    b = b/x;
```

Figure 1: An Example

While a concurrency fault is triggered by a specific interleaving among a set of memory accesses, its effects usually propagate across a single thread through data and control dependencies until it causes the program to fail [12]. When a new interleaving is introduced, it is necessary to test whether the change of a data state induced by this interleaving can introduce faults. Thus, the objective of our third step is to test propagation effects of new interleavings. To do this, TACO employs a forward static slicing technique to compute elements affected by the new interleavings and then directs concolic testing to generate new test inputs that cover them. Our preliminary results from an ongoing study demonstrate the potential of TACO in terms of cost-effectiveness.

2. PROBLEM STATEMENT

We use an example – see Figure 1, to motivate TACO as well as illustrate the novelty of TACO by comparing it to existing work. The program has two threads $T1$ and $T2$, where `g1` and `g2` are shared between the threads, and `a` and `b` are inputs to the program. Deleted lines are denoted as "--". The original and modified programs are denoted by P and P' respectively. This change in P' can cause an order violation – one type of concurrency faults, because `g1` at line 6 is no longer synchronized by wait-signal: $T2$ should not read `g1` at line 6 until a signal is called (line 14). This fault leads to a "division by zero" failure.

We now illustrate how to apply TACO to expose this regression fault. Suppose there are two existing test inputs for P, where t_1 is {`a` =2, `b` = 2} and t_2 is {`a` = 3, `b` = -1}. TACO first employs our previous work SIMRT to select test inputs that are likely to expose concurrency faults. By using impact analysis in SIMRT, `g1` at line 6 is considered affected because it is no longer protected in P', whereas `g2` is not affected because the code change does not affect the interleaving space related to `g2`. Since a concurrency fault involves at least two shared variable (SV) accesses, SIMRT identifies a set of affected SV pairs (SVPs), such that for two SVs in each pair, at least one of them is affected, and at least one of them is a write access. In this example, the affected SVPs are $svp_{\texttt{aff}_1}$ = <(6, $w(g1)$), (10, $r(g1)$)> and $svp_{\texttt{aff}_2}$ = <(6 $w(g1)$), (13, $r(g1)$)>, where w is a write and r is a read. Therefore, $t1$ is selected for reuse; $t2$ is discarded because it does not exercise affected SVPs.

In the first step of our approach, TACO takes $t1$ as a starting point. Suppose $t1$ is executed on P' following a random schedule σ_1: (8, $lock(lk)$)→(9, br)→(10, $r(g1)$)→(11, $r(g2)$)→(14, $signal(cond)$)→(15 $unlock(lk)$)→(16, br)→(1, $lock(lk)$)→(2, br)→(6, $w(g1)$)→(7, $unlock(lk)$), where br indicates a branch predicate. $svp_{\texttt{aff}_1}$ is covered in this trace. Next, given an interleaving coverage criterion, TACO seeks alternative interleaving involving $svp_{\texttt{aff}_1}$ for t_1. Suppose an inter-thread def-use criterion [6] is used, the coverage target svp_t becomes <(6, $w(g1)$)→(10, $r(g1)$)>. So TACO will need to reverse the two accesses related to `g1` in the original trace to form a new schedule σ_2 that makes (10, $r(g1)$) happen before (6, $w(g1)$). However, after σ_2 is enforced on P', no faults are detected.

TACO then proceeds to the second step. Since `g1` in $svp_{\texttt{aff}_2}$ at line 13 is not covered by $t1$ with either σ_1 or σ_2, TACO attempts to generate a new input targeting this SV. As such, t3 = {`a` =-1, `b` = 2} is generated for this purpose. By executing $t3$ on P' with a random schedule σ_3, a trace is obtained in which (13, $r(g1)$) happens before (6, $w(g1)$). So an alternative interleaving σ_4 = <(6, $w(g1)$)→(13, $r(g1)$)> becomes the next target. However, after σ_4 is enforced using t_3, the fault is still not revealed.

At the last step, for each new interleaving within its input– <$t1$, σ_2>, <$t3$, σ_4>, TACO locates the variable in a thread which is last modified by a remote access. For σ_2, `x` is modified at line 10 and that for σ_4, `x` is modified at line 13. TACO then performs a forward static slicing on $T2$, treating `x` at line 10 and line 13 as slicing criterion, so `x` at 17 becomes a new coverage target. Next, TACO invokes concolic testing to generate new inputs with respect to σ_2 and σ_4. As a result, a new input $t4$ = {`a` =1, `b` = 4} for σ_2 and an input $t5$ = {`a` =-1, `b` = 4} for σ_4 are generated. The fault is triggered by enforcing σ_4 with input $t5$.

This fault is interesting because it requires combination of a specific schedule with a particular input to manifest. This example further provides evidence showing that existing regression testing techniques for concurrent programs are not adequate. The work that is most related to TACO is our previous work SIMRT [11], CAPP [4] and recent work RECONTEST [8]. SIMRT [11] selects test cases at the input level and does not consider change impact in the interleaving space. CAPP systematically explores program changes related to concurrency semantics. RECONTEST improves efficiency of CAPP by searching problematic interleavings among only affected shared variables. However, all above techniques rely on existing test inputs and thereby restrict their capabilities in Step 1 of TACO. In contrast, TACO explores affected interleavings as well as generating new test inputs to drive program through *additional* intelreaving space induced by code changes. While full concolic testing techniques for concurrent programs such as CON2COLIC [1] considers both inputs and interleavings, they focus on single program versions and do not reduce cost of regression testing (same to re-test all). In this example, CON2COLIC would unnecessarily explore unaffected interleavings related to `g2`.

3. APPROACH

3.1 Identify Affected Elements

A key step in regression testing is to identify affected program entities induced by code changes. Our previous work SIMRT as well as RECONTEST have developed impact analysis techniques specific to concurrent programs to identify affected shared memory accesses. RECONTEST employs a dynamic change impact analysis and may miss accesses not exercised by existing test inputs. In contrast, TACO leverages SIMRT by using a conservative static analysis to identify affected shared variable pairs (SVPs) across the whole program; false positives can be eliminated during test execution. TACO also extends SIMRT by considering more scenarios, such as SVs involving user-defined synchronizations.

The set of affected SVPs, denoted as $SVP_{\texttt{aff}}$, are used to construct new interleavings, where $svp(m)_{aff}$ denotes an affected pair in $SVP_{\texttt{aff}}$ accessing to memory m. Specifically, TACO executes a test input t on P' and obtains a runtime trace $TR(t)$ that captures program execution as a sequence of events e, including memory accesses, synchronization operations, branches, and path conditions. A thread schedule σ_m = <e_m> describes a partial order relation on a set of SV accesses e_m to memory m. This order relies on a specific interleaving coverage criterion (e.g., Def-Use,

PInv) [6]. An interleaving criterion is a patten of inter-thread dependencies through SV accesses, which helps select representative interleavings to effectively expose concurrency faults. An interleaving criterion is satisfied if all feasible interleavings of SV defined in the criteria is covered. The current implementation of TACO employs a Def-Use criteria, 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 variable read by `r` between them.

An affected thread interleaving $\sigma_{\Delta_m}(t)$ contains at least one $svp(m)_{aff}$ in P'. $\overline{\sigma}_{\Delta_m}(t)$ denotes a set of all alternative interleavings of $\sigma_{\Delta_m}(t)$ by reshuffling the order of occurrences of the memory accesses m to match the interleaving coverage criterion. The interleaving space of an input, denoted as $IS(t)$, is a set of all *feasible* interleavings for all SVPs observed from $TR(t)$. $IS(t)_\Delta$ contains at least one affected interleaving. Since the interleaving prediction is performed off-line, many affected interleavings can be infeasible, which can be eliminated during test execution.

3.2 Test Suite Augmentation

Algorithm 1 displays TACO augmentation algorithm. The algorithm begins with an initial set of existing test inputs selected by SIMRT, and an iteration limit n_{iter} – a "tuning" parameter. The algorithm initializes the coverage targets $SVP_{\mathtt{t}}$ to $SVP_{\mathtt{aff}}$ – a set of affected SVP. The main loop continues until we explore all affected interleavings and their impacts, or reach the iteration limit.

Step 1. This step invokes `SchedExplorer` to explore affected interleaving space for each existing test input (line 5-7). `SchedExplorer` exercises test input t on P' with a random schedule to obtain $TR(t)$ (line 22). $\overline{IS}(t)_\Delta$ is set to empty as no alternative schedules have been explored at this point (line 23). All affected interleavings in the trace are added to $IS(t)_\Delta$ if it has not been explored by other test inputs (line 25).

Next, the algorithm invokes `SchedDiscover` to seek all alternative interleavings $\overline{IS}(t)_\Delta$ according to an interleaving coverage criterion C (line 26). The algorithm iteratively removes an interleaving σ_m from $\overline{IS}(t)_\Delta$, attempting to enforce it on P' to obtain a new trace $TR(t)'$ (line 30). If such interleaving is successfully enforced, it is added to $IS(t)_\Delta$; coverage information of affected SVP is updated (lines 31-33). During the enforced execution, we might observe affected SVPs that have not been observed so far due to the fact that the new execution changed program control flow. If such SVP is discovered, the algorithm predicts all alternative interleavings related to it and add them to $\overline{IS}(t)_\Delta$ (lines 34-35). The algorithm continues exploration until $\overline{IS}(t)_\Delta$ is empty.

Step 2. This step directs concolic testing on P' to generate new inputs to cover uncovered SVP targets in Step1. The algorithm enters a loop in which it selects each uncovered single SV in the $SVP_{\mathtt{aff}}$ serving as a target for concolic testing procedure `InputGenerator` (line 9). The parameter "-" indicates that no specific interleavings are applied. Procedural `InputGenerator` begins by locating branches (b_t) for which the source node is a predicate node p that is covered by at least one existing test inputs (lines 41-42); these become immediate targets for input generation. The algorithm then collects all path conditions PC_{b_t} for test inputs whose execution traces reach p (lines 43-44). For each such path condition pc, the algorithm generates a new path condition pc' by negating b_t in pc and removing all subsequent branches (lines 45-46). If pc' has not been seen before, and that pc' has a solution, the algorithm uses it to generate a new test case t_{new} (lines 47–49). Otherwise, the algorithm ignores it and moves on to the next path condition. After `InputGenerator` returns a new test input to the main procedural (line 9), the algorithm invokes `SchedExplorer` to explore interleaving space of this input.

Algorithm 1 TACO algorithm

```
1:  procedure AUGMENTATION(P, P', n_iter, T)
2:      SVP_aff = ConImp(P, P')
3:      SVP_t = SVP_aff
4:      while NotDone do
5:          for each t ∈ T do                                   ▷ T can be ordered
6:              SchedExplorer(t, C)
7:          end for
8:          for each sv_t ∈ SVP_aff and sv_t is not covered do
9:              t_new = InputGenerator(sv_t, T, −)
10:             SchedExplorer(t_new, C)
11:         end for
12:         for each σ_m ∈ IS(t)_Δ do                           ▷ Test propagation effects
13:             S_aff = SeqImp(rd(m_last))
14:             for each s_t ∈ SVP_t and s_t is not covered do
15:                 InputGenerator(s_t, t, σ_m)
16:             end for
17:         end for
18:         return IS_Δ
19:     end while
20: end procedure

21: procedure SCHEDEXPLORER(t, C)
22:     TR(t) = Random(P', t)
23:     IS(t)_Δ = φ
24:     for each σ_{Δ_m} ∈ TR(t) and σ_{Δ_m} is not explored do
25:         IS(t)_Δ = IS(t)_Δ ∪ σ_{Δ_m}
26:         IS(t)_Δ = IS(t)_Δ ∪ SchedDiscover(σ_{Δ_m})
27:     end for
28:     while IS(t)_Δ ≠ φ do
29:         remove a σ_m from IS(t)_Δ
30:         TR(t)' = Enforce(P', t, σ_m)
31:         if successfully enforced then
32:             IS(t)_Δ = IS(t)_Δ ∪ σ'_m
33:             Update SVP_t                                    ▷ Update coverage information
34:             if a new interleaving σ'_{Δ_m} found in TR(t)' then
35:                 IS(t)_Δ = IS(t)_Δ ∪ σ_{Δ_m}
36:             end if
37:         end if
38:     end while
39: end procedure

40: procedure INPUTGENERATOR(s_t, T, σ_m)
41:     b_t = Branch(s_t)
42:     p = Predicate(s_t)
43:     T_{b_t} = all test inputs in T that reach p
44:     PC_{b_t} = path conditions obtained from executing test cases in T_{b_t}
45:     for each pc ∈ PC_{b_t} do
46:         pc' = DelNeg(pc, b_t)
47:         if pc' ∉ PC_{b_t} then
48:             t_new = Solve(pc')                              ▷ If pc' ≠ UNSAT
49:             return t_new
50:         end if
51:     end for
52: end procedure
```

In the example of Figure 1, `r(g1)` in $svp(g1)_{aff_2}$ is the target to be covered. $t1$ is selected for reuse, since its trace contains predicate p at line 9 that controls reachability of `r(g1)`. First, $t1$'s path condition, ($a >= 0 \wedge b <= 3$), is selected. `DelNeg` is applied, obtaining another path condition, ($a < 0$). By using solver to solve this path condition, a new test input $t3$ is produced (i.e., {a = -1, b = 2}), that covers the false branch of p. At the same time, more path conditions can be collected because $t3$ may exercise new paths. Next, the algorithm uses $t3$ to explore affected interleavings related to $svp(g1)_{aff_2}$.

Step 3. This step attempts to generate test inputs to cover elements in sequential program affected by new interleavings. For each affected interleaving σ_m in $IS(t)_\Delta$, the algorithm locates the variable who reads the last remote write m_{last} in σ_m; this read value may propagate across local thread all the way to the the output. The algorithm first invokes `SeqImp` (line 13), a static forward slicing [9] module to compute a sequential slice $S_{\mathtt{aff}}$, consisting of statements affected by $rd(m_{last})$ through data and control dependences (line 14). Next, the algorithm invokes `InputGenerator` (line 15) to generate a new input to cover each such statement according to its interleaving σ_m.

In the example of Figure 1, as illustrated in Section 2, `x` at 17 is a new coverage target for <$t1$, σ_2> and <$t3$, σ_4>. The path conditions for both $t1$ and $t3$ are selected to generate new inputs because both

test inputs cover the predicate of b at line 16. Consider the PC ($a < 0 \wedge b <= 3$) for $t3$, DelNeg is applied to obtain a new PC ($a < 0 \wedge b > 3$), producing a new input $t5$ (i.e., {a =-1, b = 4}). By enforcing $t3$'s interleaving σ_4 along with $t5$, the "division by zero" fault is exposed.

3.3 Preliminary Evaluation

Our algorithm was implemented on CLOUD9 symbolic execution engine[1]. We modified CLOUD9's scheduler to control execution of affected interleavings. The impact analysis (ConImp and SeqImp modules) was implemented on CODESURFER[2], a commercial static analysis tool to perform sophisticated analysis (e.g., static slicing, data-flow analysis) on C/C++ source code. As future work we intend to implement the impact analysis on LLVM to seamlessly integrate it with our framework.

In the preliminary stage of evaluation, we applied TACO on two toy programs – PROD-CONS and RWLOCK, which are shipped with CLOUD9. We modified the programs by inserting additional predicates and concurrency semantics to increase program complexity. We used CCMUTATOR [5], a mutation generator for multithreaded C/C++ applications to simulate modified program versions by injecting 18 concurrency faults (versions). We ran each version pair on TACO with randomly generated 50 test cases. TACO detected all 18 regression concurrency faults. We next disabled InputGenerator and used only existing inputs (e.g., RECONTEST). In this case, 11 faults were detected. We next employed re-test all by applying TACO on the whole program level rather than focusing on affected program entities (e.g., CON2COLIC); this increased testing time by 120x on average. While additional studies are needed, the early results are promising, demonstrating that TACO can be more cost-effective that existing techniques.

4. CONTRIBUTIONS AND VISION

We have presented an automated regression test suite augmentation framework, TACO, for use in detecting concurrency faults that are induced due to code changes. The novelty of TACO is that it targets *two* essential aspects of concurrent programs – test inputs and thread interleavings. TACO treats test input generation and interleaving exploration uniformly, in which new test inputs are generated from test reuse to direct exploration of affected interleaving space that has not been covered by existing inputs. TACO is a configurable framework that allows engineers to flexibly manage its modules and parameters. For example, one can disable Step 2 and Step3 to actively explore thread interleavings within existing inputs. Also, one can disable impact analysis and let TACO generate inputs and interleavings for the whole program. In addition, TACO can also be used to *replay* regression concurrency faults by leveraging its active scheduler.

Our work is still preliminary and contains a few limitations. To create effective RTA techniques we need to investigate factors that can potentially influence their the cost and effectiveness. We would like to provide at lease three major factors for future research.

Interleaving coverage criteria. Interleaving coverage criteria may impact how well TACO works. Lu et. all [6] 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. [3] further confirmed that effectiveness concurrency fault detection can vary across different criteria. While TACO employs Def-Use criteria by default, it is important to investigate cost-effectiveness of RTA when using other interleaving criteria in the context of regression testing.

[1] https://sites.google.com/site/dslabepfl/proj/cloud9

[2] http://www.grammatech.com/research/technologies/codesurfer

Ordering affected program entities. TACO operates on lists of affected interleavings and sequential elements. Our current implementation employs random search for these entities, but we believe that the order in which these elements are searched can affect the techniques. We conjecture that by guiding concolic testing toward paths that cover the most not-yet-covered affected SVs in a depth-first order may speed up the augmentation process, because test cases generated for SVs earlier in flow may incidentally cover SVs occurring later in flow, obviating the need to consider those SVs again. By doing this can also maximize the number of interleavings to be explored per test input so as to reduce the number of invocations of concolic testing for later test inputs.

Test inputs effectiveness. As noted in Section 3, TACO may encounter the same affected SVPs across different inputs. To reduce cost, we employ a heuristic to test interleavings for each SVP only once. This may end up with using only earlier test inputs while ignoring later ones that may be more powerful at revealing faults, which may in turn cause concolic testing in Step 3 to be invoked more frequently than necessary. We propose three possible solutions. First, we conjecture that reordering test inputs for each iteration in the main algorithm can take advantage of diverse test inputs and may improve effectiveness of Step 2. A second solution is to leverage dynamic concurrency fault detection techniques [2] to report faults based on access patterns instead of output. This disables Step 3 but at the cost of dynamic analysis and potential false positives induced [11]. Third, we hypothesize that the depth in which predicted interleavings are actively tested for the same input can influence cost-effectiveness of the technique. Instead of testing for all predicted interleavings within the same input, we can let TACO use "round-robin" across test inputs in a manner that after an interleaving is explored, TACO proceeds to a different input until all inputs are used, then starting again from the first input to explore a different interleaving and so on.

5. REFERENCES

[1] A. Farzan, A. Holzer, N. Razavi, and H. Veith. Con2Colic Testing. In *FSE*, pages 37–47, 2013.

[2] C. Flanagan and S. N. Freund. FastTrack: Efficient and Precise Dynamic Race Detection. In *PLDI*, pages 121–133, 2009.

[3] 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.

[4] V. Jagannath, Q. Luo, and D. Marinov. Change-aware Preemption Prioritization. In *ISSTA*, pages 133–143, 2011.

[5] M. Kusano and C. Wang. CCmutator: A Mutation Generator for Concurrency Constructs in multithreaded C/C++ Applications. In *ASE*, pages 722–725, 2013.

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

[7] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and Reproducing Heisenbugs in Concurrent Programs. In *OSDI*, pages 267–280, 2008.

[8] V. Terragni, S.-C. Cheung, and C. Zhang. RECONTEST: Effective Regression Testing of Concurrent Programs. In *ICSE*, 2015.

[9] M. Weiser. Program Slicing. In *ICSE*, pages 439–449, 1981.

[10] Z. Xu, Y. Kim, M. Kim, G. Rothermel, and M. B. Cohen. Directed Test Suite Augmentation: Techniques and Tradeoffs. In *FSE*, pages 257–266, 2010.

[11] T. Yu, W. Srisa-an, and G. Rothermel. SimRT: An Automated Framework to Support Regression Testing for Data Races. In *ICSE*, pages 48–59, 2014.

[12] W. Zhang, J. Lim, R. Olichandran, J. Scherpelz, G. Jin, S. Lu, and T. Reps. ConSeq: Detecting Concurrency Bugs Through Sequential Errors. In *ASPLOS*, pages 251–264, 2011.