CONCURRIT: Testing Concurrent Programs with Programmable State-Space Exploration

Jacob Burnim Tayfun Elmas George Necula Koushik Sen
Department of Electrical Engineering and Computer Sciences, University of California, Berkeley
{jburnim,elmas,necula,ksen}@cs.berkeley.edu

Abstract

Testing is the most widely-used methodology for software validation. However, due to the nondeterministic interleavings of threads, traditional testing for concurrent programs is not as effective as for sequential programs. To attack the nondeterminism problem, software model checking techniques have been used to systematically enumerate all possible thread schedules of a test program. But such systematic and exhaustive exploration is typically too time-consuming for many test programs. We believe that the programmer's help to guide the model checker towards interesting executions is critical to circumvent this problem.

We propose a testing technique and a supporting tool called CONCURRIT, which provides a model checker that can be guided programmatically within test code. While writing a test, the programmer specifies a particular thread interleaving scenario in mind using an embedded domain-specific language (DSL), and CONCURRIT explores all and only the executions realizing the intended scenario. During the exploration, the programmer is also able to observe the execution (e.g., assert invariants) and constrain the future decisions of the model checker, all within the test code. We believe that providing the programmer the ability to observe and control the exploration of executions will lead to more effective and efficient testing for concurrent programs.

1. Introduction

Testing is the most widely-used methodology for software validation. Concurrency brings new challenges to software testing. The biggest challenge is the nondeterminism in the scheduling of concurrently running computations, i.e., threads. The output of a concurrent program can be highly sensitive to timings of the interactions and interference between threads. The same test program can result in a large number of nondeterministic executions producing different outcomes. This makes concurrency-related defects notoriously difficult to detect and reproduce. As a result, testing has not been applied on concurrent programs as effectively in practice as it has on sequential programs.

We believe that to address this problem, it is critical for testing to provide the programmer techniques and tools (i) to conveniently express interesting executions of a test program with respect to a particular thread interleaving scenario in mind and (ii) to examine these executions systematically and efficiently. We observe a spectrum of approaches to this, as described next.

1.1 Background

At one end of the spectrum, there are manual approaches. In the extreme case, a multithreaded test program is run under a completely nondeterministic scheduler. There is no guarantee about whether the resulting execution will be of any interest (e.g., with sufficient degree of interaction between threads) to the programmer and whether re-running the test will produce the same or different executions. To control the nondeterminism, the programmer implements a synchronization/communication mechanism (e.g., inserts sleep statements throughout the code) to restrict the possible schedules of threads towards a particular scenario. However, such mechanisms are not portable and reliable in general and may require nontrivial modifications in the program text. Techniques have been proposed to guide the execution to intended thread schedules in more portable and reliable ways, where the intended schedules are specified by the programmer relative to a global timer (ConAn [16], MultithreadedTC [21]) or a sequence of user-defined events expressed in linear temporal logic (IMUnit [10]). While these techniques give the programmer the ability to impose constraints on the interleaving of threads, they do not support systematic exploration of all executions satisfying these constraints.

At the other end of the spectrum, there are fully automated approaches. To alleviate the nondeterminism in the execution, software model checking techniques have been combined with testing to control the thread scheduler so that distinct interleavings of the threads in the test are systematically enumerated and checked

against the test criteria [14, 19, 20]. Many model checking exploration techniques have been developed that seek to achieve high coverage of executions in a scalable way [12]—such as partial-order reduction [7, 8], symmetry reduction [9], thread-modular reasoning [6], and preemption bounding [19]. These techniques do not directly help to efficiently examine interesting and potentially-problematic interleaving scenarios. Due to the state-space explosion problem, examining a particular scenario involving large components of the program and identifying a bug may require to wait for an expensive (in time and resources) model checking process.

Recent studies proposed techniques to control the scheduling of a model checker to target suspicious executions more quickly than traditional state-space exploration. Among them, active testing [22], probabilistic scheduling [2], and change-aware preemption prioritization [11] are fully automated and rely on heuristics. However, we believe that programmers help to guide the exploration of executions is also valuable, and testing tools should be designed to allow the programmer to interact with the test runtime to express her intents and insights about the test scenario. In fact, work on preemption sealing [1] proposed to disable preemptions that the programmer thinks are not interesting or can cause false warnings. In our work, we would like to give more control and flexibility to the programmer in this direction.

1.2 Our Approach

We propose a modeling and implementation of a testing technique and supporting tool (CONCURRIT) that combines testing and software model checking in a novel way. Our main goal is to make the model checking more accessible and controllable to the programmer. In particular, we provide a domain-specific language (DSL) embedded in a host language (in our case C/C++) using which the programmer can specify at high level how an execution of the test should develop, indicating constraints and nondeterministic choices on thread schedules. For example, a constraint on the schedule may dictate which threads are allowed to be interleaved at which point of the execution, or at which code location or under what conditions control will pass from a thread to another. Our DSL provides to the programmer a concise and convenient way of imposing constrains on thread interleavings programmatically—i.e., within the test code. In this way, one can describe a test scenario with various degrees of flexibility in the thread schedule ranging from fully deterministic to fully nondeterministic. Section 2 gives example test programs written in our DSL. CONCURRIT implements a testing framework and a stateless model checker that, guided by our DSL, enumerates all possible executions satisfying the scenario constraints.

Our approach also makes the underlying model checker more accessible to the programmer for implementing and evaluating a custom search strategy within the test code. While existing model checkers also offer plug-in mechanisms to customize and extend the search algorithms, we believe that interacting with the tool directly from the test code and using a high-level DSL (rather than a low-level API) is more convenient for programmers. In this aspect, our work is similar to [15], which separates a compact and high-level fixed-point formulation of a model checking algorithm for Boolean programs from a general-purpose fixed-point solver.

The initial prototype implementation of CONCURRIT for C/C++ programs is available online at `http://code.google.com/p/concurrit/`.

1.3 Testing Programs in Cooperative Semantics

One of our key insights is that from the perspective of a programmer, cooperative executions of a program are relatively much simpler to describe and reason about than its traditional preemptive executions. Thus, we propose to test a concurrent program by only examining its cooperative executions. In this case, code locations that a thread may yield control to another thread are explicitly marked, and a thread may not release control at a different location. In other words, code between two yield locations are guaranteed to run atomically. Once the functionality of the program is tested in the cooperative semantics, there are techniques (e.g., Yi et al. [23–25]) that can validate the choice of yield points by checking that the program is *cooperative* [25] with these yield points—i.e., when run under a traditional preemptive scheduler, the program does not exhibit any extra behavior other than it exhibits when running under a cooperative scheduler.

Let P refer to the program under test. We propose to perform the validation of P in the following steps:

S1 The programmer obtains from P a program P_{coop} by annotating P with yield points. (Section 3.1)

S2 The programmer writes multithreaded tests for P_{coop} and runs them in the cooperative semantics. This paper is mainly about the testing of P_{coop} in this step.

S3 The programmer (using [24] or [25]), either (i) shows that P_{coop} is cooperative, or (ii) identifies potential interference points that are not covered by the existing yield annotations in P_{coop}. If (ii) happens, back in S1, the programmer adds yield points to cover detected interference and re-runs the tests in S2.

S4 The programmer compiles and deploys the program in the preemptive semantics by simply ignoring the yield annotations. In our case, yield annotations are C macros that translate to no-ops in the release build.

```
1  // Program code under test (annotated with yield points for testing)
2  producer(Buffer *buff, int product) { ... ... ... }
3  consumer(Buffer *buff) {
4    ...
5    if (buff is empty) { ... yield("Await"); ... }
6    ...
7  }
8  ------------------------------------------------------------------------------
9  // Test script in our DSL describing all interleavings of four threads
10 test_producer_consumer_1() {
11   Buffer buff(2);                  // Create empty bounded buffer of size two

12   // Create producer and consumer threads, but not activate them yet
13   thread_t P1 = thread(producer, &buff, 1),  P2 = thread(producer, &buff, 2);
14   thread_t C1 = thread(consumer, &buff),  C2 = thread(consumer, &buff);

15   while(!all_ended());                // Loop until both threads terminate
16     transfer(*).until(*);             // Run a thread until some yield point
17     assert( ... );                    // Check an invariant on the buffer
18   }
19   assert(buff.size() == 0);           // Check a post-condition on the buffer
20 }
21 ------------------------------------------------------------------------------
22 // Test script in our DSL describing intended interleavings of a scenario
23 test_producer_consumer_2() {

24   ... // Same definitions in lines 11-14 above

25   // T1
26   transfer(C1).until("Await");   // Run C1 until yield point labeled "Await"

27   // T2
28   with(P1, P2, C2) {       // Execute lines 29-31 with only P1, P2 and C2
29     while(!ended(C2));                // Loop until C2 terminates
30       transfer(*).until(*);           // Run a thread until some yield point
31     }
32   }
33   assert(buff.size() <= 1);           // Check a condition on the buffer

34   // T3
35   while(!all_ended()) {               // Loop until all threads terminate
36     transfer(*).until(end);           // Run a thread until it terminates
37   }
38   assert(buff.size() == 0);           // Check a condition on the buffer
39 }
```

Figure 1. Example producer-consumer tests.

2. Example: A Producer-Consumer Test

Figure 1 shows two tests for checking a concurrent producer-consumer module, which uses a shared, bounded buffer of integers. We omit the details of the program under test (shown at the top of the figure), except we highlight that the code is annotated with yield points each with a unique label. The figure shows only one of these annotations labeled "Await" (line 5), though others exists so that threads running producer and consumer can interleave with each other. We focus on the test procedures, we call *test scripts*, written using our DSL. In the code, keywords specific to our DSL are shown underlined. For simplicity of exposition, we use a conceptual language, whereas in our implementation, the DSL is embedded in C/C++, and the corresponding language constructs are provided as C macros that translate to API calls to the CONCURRIT library.

The first test test_producer_consumer_1 describes a scenario involving a bounded buffer to hold 2 integers (line 11), two threads P1, P2 to act as producers to insert 1 and 2 into the buffer (line 13), and two threads C1, C2 to act as consumers (line 14). Using our DSL, the programmer specifies that she expects to examine all possible nondeterministic interleavings of the threads. The DSL expression all_ended() returns *true* iff all the threads (visible in the current scope) have terminated. The statement transfer(∗).until(∗) at line 16 instructs the model checker to run one of the threads P1, P2, C1, C2 until it reaches some yield point; ∗'s indicate that both the thread and yield point are chosen nondeterministically. The transfer statement blocks during this execution. We point out that, in addition to checking a post-condition of the test (line 19), the programmer can also insert an assertion at each interleaving point to check a program invariant (line 17). In this way, the programmer avoids to add these checks inside the program code under test, and thus, decouples her program from its tests.

The real novelty of our DSL comes into play when the programmer is interested in a particular set of interleavings rather than a completely nondeterministic one. The second test test_producer_consumer_2 describes such a situation. Suppose that the programmer does not want to examine all interleavings of these producer and consumer threads, because she suspects that a concurrency error is likely to occur during a particular interleaving scenario, and she wants to ensure that the intended scenario does not trigger any errors. Note that, looking at all interleavings of the threads (i.e., without constraining the interleavings) would help the programmer check her claim about the error-freedom of that interleaving scenario. However, exploring all interleavings would be too costly in time. Thus, the programmer intends to check all and only the possible executions of the scenario in mind, which develops in three steps:

T1 Thread C1 runs first until it detects that the buffer is empty and starts waiting for the buffer to get full.

T2 Then, both producer threads and consumer thread C2 run concurrently, until C2 consumes the integer either P1 or P2 inserts in the buffer and finishes. At the end of this step, size of the buffer must be $\leqslant 1$, since C2 consumes one of the integers produced.

T3 Finally, both producer threads and C1 run sequentially in some arbitrary order. At the end, the buffer must be empty, since in order to terminate C1 must consume the remaining integer not consumed by C2.

As shown in Figure 1, our DSL provides a clean, concise, and high-level way describing the scenario above. First, the DSL constructs help the programmer to constrain the nondeterminism in the interleavings of producer and consumer threads, so that at each point only allowed threads can run (using with), and a thread cannot execute earlier than or beyond a given point (using

until). For example, C1 should run up to a yield point labeled "Await" at the beginning of the test (line 26), and then it should not run until C2 ends (line 28-32). (In fact, one can describe in this style a fully deterministic interleaving indicating which threads to transfer at each point and until which yield point.) Second, the DSL constructs also allow the programmer to indicate expected nondeterminism about which thread to choose and until when that thread must run (using $*$). For example, in T2, P1, P2, and C2 are interleaved with each other nondeterministically, and in T3, threads are run sequentially until termination (we use the special label end to refer to the end of a thread) but in a nondeterministic order. Therefore, this test script specifies not a single execution but a set of executions, which our integrated model checker targets to systematically enumerate. We note that various model checking techniques, e.g., dynamic partial order reduction [5], can still be applied in this setting to make the exploration efficient.

2.1 When a Test Passes or Fails

While testing a sequential program, the common question asked by the programmer is: *Does the (only) execution of the test satisfy all the assertions?* Having the ability to search all executions of a concurrent test program, we can talk about two questions to answer when running a test with a model checker. Each question determines when the overall test passes or fails (in other words, when the model checker finishes the exploration of executions). A *successful* execution of the test script is a "terminating" execution that (i) does not violate an assertion and (ii) satisfies all the constraints specified by the programmer. Section 3.2.1 overviews our DSL constructs to impose constraints on executions. A *failing* execution is one that violates an assertion. If an assertion violation is detected, the model checker terminates the exploration immediately and declares the test failed.

The model checker can be run one of two modes to answer the following questions, respectively:

Are there any successful executions of the test script?
In this mode, the model checker seeks to find a successful execution. The test passes if the model checker explores at least one successful execution without detecting any failing execution meanwhile. The test fails otherwise. Notice that, a test in this mode can pass, producing a successful execution, even though there also exists a failing execution.

Are there any failing executions of the test script?
In this mode, the model checker enumerates all successful executions of the test script. The test fails if the model checker detects a failing execution during the exploration and passes otherwise. Differently from the previous mode, a test in this mode can pass even though there is no successful execution.

3. Testing Programs with CONCURRIT

3.1 Annotations for Cooperative Execution

Cooperative concurrency builds on the idea of (symmetric) *coroutines* [4, 13], which generalize subroutines to allow multiple entry points for suspending and resuming the execution. In CONCURRIT, we model coroutines for C/C++ using pthreads [18] and cooperation at yield points using proper synchronization operations. In our case, the cooperation among threads is performed explicitly using two kinds of operations: transfer and yield. The former is used in the test script to pass control to other threads. The latter is used for a thread (for example, P1, P2, C1, C2 in Figure 1) to relinquish the control back to the test script, and for this, the program's code under test needs to be explicitly annotated with yield before testing, as explained next.

Every yield annotation is given a unique label, using which the yield location can be referred to in the test script. Let l range over labels. A yield(l) statement can be placed anywhere in the code to indicate a location at which a thread executing that code in a cooperative execution may yield control to other threads. Figure 1 shows an example to this at line 5. The yield points in general indicate source locations, such as accesses to shared memory, that are subject to interference by other threads. Thus, similarly to [23], yield points can also be annotated as follows: A read from a shared variable x can be replaced by yield_read(l, x). This expresses that the current thread first yields the execution to other threads, and when the control is transferred back to it again, it reads from x and returns the value. A write to a variable x in the form of $x = e$ can be replaced by yield_write$(l, x) = e$. This is similar to yield_read, except that the current thread writes the (previously computed) value of e to x after it gains the control back.

In the extreme case, one can treat every shared variable access a potential yield point and imagine a tool that examines shared variable accesses and automatically inserts yield annotations accordingly. We provide such a tool (using Pin [17]) that monitors sample executions of the test program and marks accesses that involve in a data race as potential yield points. However, in reality the number of shared variable accesses subject to harmful interference may be less than all shared accesses [3, 25]. In this case, the programmer can choose to add the annotations gradually (for example, by pruning out the yield annotations inserted by a tool) as more tests pass. In fact, we see the gradual, test-driven addition of yield annotations as a systematic way of increasing the concurrency of the program.

3.2 Test Scripts: Syntax and Semantics

Figure 2 shows our DSL for writing tests. Note that, for simplicity we present here a conceptual language rather

$f, g \in Functions \quad t \in ThreadVariables$

l	::= next | end | "label text"	*Yield labels*
e	::= ended (t) | all_ended() $\mid \cdots$	*Boolean expressions*
t	::= $* \mid t \mid t_1, \cdots, t_k$	*Thread expressions*
u	::= $* \mid l \mid e$	*Until expressions*
c	::= except(**t**) | until(**u**) | count(k)	*Transfer clauses*
s	::= t = thread (*f*, *args*)	*Thread creation*
	| t = transfer (**t**).$\mathbf{c}_1 \ldots \mathbf{c}_k$	*Transfer to a thread*
	| assert (e)	*Assertion*
	| assume (e)	*Assumption*
	| with/without $(t_1, \cdots, t_k)$ { **s** }	*Thread scope*
	| $x = e$ | **s** ; **s** | if(e) { **s** } $\mid \cdots$	*Standard C/C++ stmts.*

Figure 2. Syntax of our DSL, embedded in C/C++, for writing test scripts. (We do not show yield, as it is not used in test scripts, but in the program under test.)

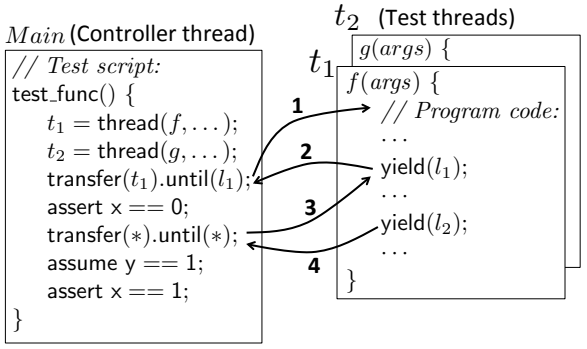

Figure 3. Execution of a test script.

than the actual C/C++ syntax. A *test script* in this setting is a function (in a recognizable signature by our testing framework), whose body is a statement **s** from Figure 2. A test script combines our DSL expressions and statements with other, standard expressions and statements of the host language (in our case, C/C++).

Figure 3 depicts the execution of a test. The test is executed by a set of threads: $\{Main, t_1, ..., t_n\}$ whose interleavings are controlled by a model checker. $Main$ refers to the special controller thread running the test script written in our DSL. In Figure 1 the body of test_producer_consumer_1/2 are executed by $Main$. Every execution of the test starts with $Main$ being the only thread. Other threads $(t_1, ..., t_n)$, we call *test threads*, are created by $Main$ within the script using the thread operation; each test thread executes a given function call concurrently (and cooperatively) with $Main$ and other test threads. In Figure 1 threads P1, P2, C1, C2 are test threads and created at lines 13-14. Creating a thread does not activate the new thread; the thread is activated later by a transfer statement as described next.

At any time during the execution, either $Main$ or one of $t_1, ..., t_n$ is allowed to run, and that thread does not pass the control to another thread until it reaches a transfer or yield.

- Control passes from the $Main$ thread to a test thread t_i by a transfer operation in the test script. (Arrows 1, 3 in Figure 3) As only one thread is active at any time, $Main$ then gets into a waiting state.
- Control passes from a test thread t_i to the $Main$ thread by a yield operation. (Arrows 2, 4 in Figure 3) Not every yield call is granted (taken) (see Section 3.2.1), but whenever it is, control is always given to $Main$.

When $Main$ becomes active, it can check some condition on the current state using assert or assume (see below), perform local computation and update some (possibly global) variables, and/or can transfer control to another test thread. An execution of the test ends when the test script finishes. The model checker may then backtrack and start a new execution of the script.

We point out that the test script may also describe an infeasible interleaving. For example, a thread may block (on a synchronization operation) between two yield point, preventing $Main$ from taking the control back. To detect and avoid such cases, we set a time limit for each thread to execute between two yield points. If this time expires, the model checker stops the current execution and backtracks for a distinct execution.

3.2.1 DSL Constructs to Constrain Test Executions

Assert and assume. Given a boolean expression on the current state (global variables plus the locals of $Main$) we distinguish two ways to reason about a condition on the current program state: assert e and assume e. Both statements do not have any side effects if e holds (i.e., evaluates to *true*), but they differ when e does not hold.

If e does not hold, assert e causes the test to fail immediately, as usual. On the other hand, assume e in this case does not terminate the entire test. Instead, the model checker discards the currently explored execution, backtracks, and restarts with a distinct thread schedule. Thus, we use assumptions to impose a "soft" constraint on the executions to be explored by the model checker, especially when expressing at which conditions an interleaving is expected to happen is more convenient than referring to a particular yield label in the program.

Nondeterministic transfers. While a test thread always yields to $Main$, the target of a transfer from $Main$ can be decided statically or at runtime. In the simplest case, transfer(t_i) passes the control to thread t_i. The programmer can also leave the (nondeterministic) decision about which thread to run to the model checker using transfer$(t_1, ..., t_k)$ or transfer$(*)$. In the former case one of $t_1, ..., t_k$ and in the latter case any thread in the cur-

rent scope (see below) is chosen.[1] The chosen target thread is returned by transfer for the script's observation when control is given back to $Main$. An except clause can be attached to transfer$(*)$ to prevent a thread or threads from being scheduled at that transfer point (e.g., transfer$(*)$.except(t_2)).

Transfer clauses. When transferring to a test thread, the programmer can also choose at which yield point the target thread should yield the control back to $Main$. For this, transfer statements are augmented with count and until clauses, which specify the required conditions for the target thread to yield. By default, each transfer is augmented with until$(*)$ and count(1) clauses, indicating that the target thread can relinquish the execution at any point.[1] Given a label l, until(l).count(k) indicates that the target thread should yield at the k^{th} visit of a yield point labeled with l but no earlier. We also define special labels next and end for convenience to refer to the next reached yield label and the end of the target thread. For example, transfer(t).until(end) transfers to t and executes it until t terminates. Finally, the programmer can also specify a boolean expression in until(e) to indicate that the target thread should yield only when e evaluates to *true*. Given that, transfer$(\cdots)$.until(e) is equivalent to transfer$(\cdots)$; assume(e).

Thread scopes. Our DSL allows the programmer to define a temporary *thread scope* in which the model checker works with only a subset of threads visible in the current C scope. For this, we provide with$(t_1, \cdots, t_k)$ and without$(t_1, \cdots, t_k)$ statements. The former defines a thread scope in which only threads $t_1, \cdots, t_k$ are used in any scheduling decision while executing the given statement (referring to another thread causes an error). In Figure 1, we use with to constrain part of the execution to threads P1, P2 and C2 (line 28). The latter also defines a thread scope but in a dual way; it temporarily removes $t_1, \cdots, t_k$ from the current scope.

4. Conclusion

We propose a modeling and implementation of a testing technique and supporting tool (CONCURRIT) combining testing and software model checking in a novel way. Our technique provides a domain-specific language (DSL) embedded in a host language (in our case C/C++). The programmer, using this DSL, can specify at high level how an execution of the test should develop; s/he can explicitly indicate constraints on thread schedules. In this way, the programmer can control the degree of freedom in nondeterministic choices a model checker would make during the state-space exploration. We believe that such a DSL can make the model checking more accessible and controllable to the programmer, and thus, can lead to a more effective use of the state-space exploration in unit and system testing.

[1] The nondeterministic decision may be biased by the model checker's search algorithm, e.g., the partial-order reduction [7] being used.

References

[1] Thomas Ball, Sebastian Burckhardt, Katherine Coons, Madanlal Musuvathi, , and Shaz Qadeer. Preemption sealing for efficient concurrency testing. *Technical Report MSR-TR-2009-143, Microsoft Research*, 2009.

[2] Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ASPLOS '10, pages 167–178, New York, NY, USA, 2010. ACM.

[3] Jacob Burnim, Tayfun Elmas, George Necula, and Koushik Sen. NDSeq: Runtime checking for nondeterministic sequential specifications of parallel correctness. In *Programming Language Design and Implementation (PLDI)*, 2011.

[4] Melvin E. Conway. Design of a separable transition-diagram compiler. *Commun. ACM*, 6:396–408, July 1963.

[5] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In *POPL '05: Proc. of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 110–121, New York, NY, USA, 2005. ACM Press.

[6] Cormac Flanagan and Shaz Qadeer. Thread-modular model checking. In *Proceedings of the 10th international conference on Model checking software*, SPIN'03, pages 213–224, Berlin, Heidelberg, 2003. Springer-Verlag.

[7] Patrice Godefroid. *Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem*, volume 1032. Springer-Verlag Inc., New York, NY, USA, 1996.

[8] Guy Gueta, Cormac Flanagan, Eran Yahav, and Mooly Sagiv. Cartesian partial-order reduction. In *Proceedings of the 14th international SPIN conference on Model checking software*, pages 95–112, Berlin, Heidelberg, 2007. Springer-Verlag.

[9] Radu Iosif. Symmetry reduction criteria for software model checking. In *Proc. of the 9th International SPIN Workshop on Model Checking of Software*, pages 22–41, London, UK, 2002. Springer-Verlag.

[10] Vilas Jagannath, Milos Gligoric, Dongyun Jin, Qingzhou Luo, Grigore Rosu, and Darko Marinov. Improved multithreaded unit testing. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ESEC/FSE '11, pages 223–233, New York, NY, USA, 2011. ACM.

[11] Vilas Jagannath, Qingzhou Luo, and Darko Marinov. Change-aware preemption prioritization. In *Proceedings of the 2011 International Symposium on Software Testing*

and Analysis, ISSTA '11, pages 133–143, New York, NY, USA, 2011. ACM.

[12] Ranjit Jhala and Rupak Majumdar. Software model checking. *ACM Comput. Surv.*, 41:21:1–21:54, October 2009.

[13] Gilles Kahn and David B. Macqueen. Coroutines and Networks of Parallel Processes. In *Information Processing 77*, pages 993–998. North Holland Publishing Company, 1977.

[14] Moonzoo Kim, Yunho Kim, and Hotae Kim. A comparative study of software model checkers as unit testing tools: An industrial case study. *IEEE Trans. Softw. Eng.*, 37:146–160, March 2011.

[15] Salvatore La Torre, Madhusudan Parthasarathy, and Gennaro Parlato. Analyzing recursive programs using a fixed-point calculus. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 211–222, New York, NY, USA, 2009. ACM.

[16] Brad Long, Daniel Hoffman, and Paul Strooper. Tool support for testing concurrent java components. *IEEE Trans. Softw. Eng.*, 29:555–566, June 2003.

[17] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[18] F. Mueller. Pthreads library interface, 1993.

[19] Madanlal Musuvathi and Shaz Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 446–455, New York, NY, USA, 2007. ACM.

[20] V. Mutilin. Concurrent testing of java components using java pathfinder. In *Leveraging Applications of Formal Methods, Verification and Validation, 2006. ISoLA 2006. Second International Symposium on*, pages 53 –59, nov. 2006.

[21] William Pugh and Nathaniel Ayewah. Unit testing concurrent software. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ASE '07, pages 513–516, New York, NY, USA, 2007. ACM.

[22] Koushik Sen. Race directed random testing of concurrent programs. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 11–21, New York, NY, USA, 2008. ACM.

[23] Jaeheon Yi, Tim Disney, Stephen N. Freund, and Cormac Flanagan. Types for precise thread interference. Technical Report UCSC-SOE-11-22, University of California at Santa Cruz, 2011.

[24] Jaeheon Yi and Cormac Flanagan. Effects for cooperable and serializable threads. In *Proceedings of the 5th ACM SIGPLAN workshop on Types in language design and implementation*, TLDI '10, pages 3–14, New York, NY, USA, 2010. ACM.

[25] Jaeheon Yi, Caitlin Sadowski, and Cormac Flanagan. Cooperative reasoning for preemptive execution. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 147–156, New York, NY, USA, 2011. ACM.