CONCURRIT: A Domain Specific Language for Reproducing Concurrency Bugs

Tayfun Elmas [†] Jacob Burnim [‡] George Necula Koushik Sen

Department of Electrical Engineering and Computer Sciences, University of California, Berkeley

{elmas, jburnim, necula, ksen}@cs.berkeley.edu

Abstract

We present CONCURRIT, a domain-specific language (DSL) for reproducing concurrency bugs. Given some partial information about the nature of a bug in an application, a programmer can write a CONCURRIT script to formally and concisely specify a set of thread schedules to explore in order to find a schedule exhibiting the bug. Further, the programmer can specify how these thread schedules should be searched to find a schedule that reproduces the bug. We implemented CONCURRIT as an embedded DSL in C++, which uses manual or automatic source instrumentation to partially control the scheduling of the software under test. Using CONCURRIT, we were able to write concise tests to reproduce concurrency bugs in a variety of benchmarks, including the Mozilla's SpiderMonkey JavaScript engine, Memcached, Apache's HTTP server, and MySQL.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

Keywords Concurrency Errors; Domain-Specific Languages; Software Testing

1. Introduction

To diagnose and fix a software bug, a programmer often needs to first reproduce the bug. That is, given some partial information about the nature of the bug — e.g., from a bug report or from directly observing erroneous program behavior — the programmer constructs a test scenario in which they execute some piece of their software so that it reliably exhibits the bugs. Reproducing such bugs can require controlling a number of sources of nondeterminism in a program's execution, including program inputs, library dependencies, and interactions with the underlying operating system. This task is especially challenging for concurrent programs, where a bug may occur only under very specific interleavings of a program's threads.

[†]The author is currently affiliated with Google, Inc., and can be reached at tayfunelmas@gmail.com.

[‡]The author is currently affiliated with Sift Science, Inc., and can be reached at jburnim@gmail.com.

PLDI'13, June 16–19, 2013, Seattle, WA, USA.
Copyright © 2013 ACM 978-1-4503-2014-6/13/06...$15.00

To reproduce concurrent bugs in practice, programmers often do *stress testing*. In stress testing, a programmer imposes *no control* over the scheduling of threads. Rather, to increase the chance of examining different and interesting schedules, the programmer creates a long-running test scenario with a large number of threads. In addition, the programmer adds perturbations to the timings of the threads by exploiting her (partial) knowledge about the bug. For example, if the programmer has intuition about potentially problematic locations in the code, she inserts *sleep* statements at these locations to perturb the thread schedule in a certain way. Such tests can be long-running and the resulting buggy executions can be very difficult to understand and debug due to the large number of threads and complex test setups. Manually controlling the schedule, e.g., by inserting sleeps, can be ad hoc and unreliable and may require nontrivial modifications in the program text.

In this paper, we propose CONCURRIT, a domain-specific language (DSL) for reproducing concurrency bugs. When using CONCURRIT to reproduce a bug, as with the existing approaches described above, a programmer begins with partial knowledge about the bug, for example, a number of threads and some intuition about how to schedule these threads to exhibit the bug. Then, the programmer also writes a *test script* in the CONCURRIT language. This CONCURRIT test script expresses the programmer's knowledge about the bug formally and concisely. In particular, the test script describes a set of potentially-problematic thread schedules among which the programmer intends to search for the target, buggy interleaving. For this, the test script dictates partial constraints on the thread schedule, for example, which threads are allowed to be interleaved at which point of the execution, or at which code location or under what conditions a context switch will happen. At runtime, the script guides the interleaving of threads, within the constraints given in the script, to generate these thread schedules.

CONCURRIT also gives the programmer the ability to implement various techniques to search for the target schedules. For example, combined with automatic or manual instrumentation of the program under test, the programmer can impose any level of control from none at all, as in random search, to full control of the thread schedule, as in model checking (e.g., [19]). For this, the programmer uses DSL constructs to explicitly indicate nondeterministic choices in the interleaving (e.g., a set of threads one of which will be scheduled next). At each execution of the test, these choices are resolved in a controlled manner, in order to search for distinct thread schedules with respect to these choices. While the choice points for the schedule are explicitly indicated in the CONCURRIT test, the underlying mechanics of the search are transparent to the programmer. For example, the search can be implemented independently from the test by employing one of a number of search strategies we provide, such as depth- and breadth-first searches. This approach also enables rapid prototyping and evaluation of a custom search strategy by writing and exploring a CONCURRIT test.

CONCURRIT allows a programmer to simply specify and explore a set of thread schedules with various degrees of flexibility. For example, the programmer may start with little intuition about a bug and write a fully nondeterministic CONCURRIT test (i.e., any thread may be scheduled any time), which results in a high number of thread schedules. As she gains more insight about the bug based on the results of the CONCURRIT test, she can refine the test to incorporate new insights, search fewer schedules, and reproduce the target bug more quickly. For example, if the programmer thinks that the bug resides in a small number of critical functions, she can modify the test to express her interest in different interleavings of threads only when the program is in one of these functions, leaving a reasonable number of interleavings to explore. Furthermore, one can write a fully deterministic test, which specifies an exact schedule triggering the bug. A CONCURRIT test specifying one or few schedules can be used to formally document the bug and for regression testing after the bug is fixed.

Searching for thread schedules in our work carries similarities to model checking techniques that enumerate thread schedules, e.g., [6, 14, 19, 21]. However, our approach diverges from the traditional model checking in two significant aspects. First, in the CONCURRIT approach, the programmer can guide the exploration of the schedules by imposing, within the test script, constraints on individual schedules. This allows the programmer to restrict and localize the search, and to focus on small number of interesting program states and interleavings, and have the search terminate earlier. Second, the traditional notion of model checking assumes that (1) the thread schedule is a primary source of nondeterminism, and (2) the entire thread schedule can be controlled. These assumptions are fundamental for the soundness guarantees of such model checking. Nevertheless, these assumptions only hold for relatively small programs, not for large, complex software, such as web browsers and servers. Controlling the entire schedule is challenging for large programs, because precisely repeating an entire execution prefix is very difficult due to sources of nondeterminism (e.g., timing of computations and network events) other than the thread schedule. Therefore, we accept that totally avoiding uncontrolled nondeterminism is not realistic in practice, and we develop our testing approach to embrace the possibility of uncontrollable nondeterminism. A CONCURRIT test can describe the controlled nondeterminism for part of the thread schedule and leave the rest of the schedule unspecified. As a result, for example, a search for distinct thread schedules with respect to the controlled nondeterministic choices in the CONCURRIT test can be performed in the presence of partially uncontrolled interleavings of threads. In fact, we do not have to control the entire thread schedule, because concurrency bugs in general can be triggered by controlling only a small (but critical) part of the schedule. From this point of view, our approach inherits the imperfect but lightweight and tolerant notion of control from sleep-based approaches, and formal notion of search from model checking.

Contributions and outline. In Section 2, we present a case study showing how to use CONCURRIT to reproduce a real bug in Mozilla's SpiderMonkey JavaScript engine. We formally present the CONCURRIT DSL, discuss how to write CONCURRIT search strategies, and present several high-level CONCURRIT constructs and patterns for writing effective and concise CONCURRIT test scripts, in Sections 3, 4, and 5, respectively. In Section 6, we present our implementation of CONCURRIT as an embedded DSL in C++. Our framework supports the use of CONCURRIT for both unit testing within a single process and system testing involving multiple processes. In Section 7, we describe our experimental evaluation of CONCURRIT, on a number of of multithreaded programs from well-known Inspect [26], PARSEC [3], and RADBench [12] benchmark suites. Our benchmarks include large software such as the Mozilla's SpiderMonkey JavaScript engine, Memcached, the Apache HTTP server, and MySQL. We demonstrate the expressiveness and usefulness of CONCURRIT by writing tests to reproduce concurrency bugs in these benchmarks. Our implementation is available at `http://code.google.com/p/concurrit/`.

2. Overview: Writing tests in CONCURRIT

We present a case study of using CONCURRIT from the perspective of a programmer who wants to write a test to reproduce a concurrency bug. We consider a real concurrency bug in the Mozilla SpiderMonkey JavaScript Engine (version 1.8RC1), a large software system with 121K lines of code. (In Section 7 this bug is named *spidermonkey2*.) The related bug report can be found at: `https://bugzilla.mozilla.org/show_bug.cgi?id=478336`. The bug manifests itself as an assertion violation and can be triggered using the multithreaded program given in Figure 1 (supplied with the bug report). We refer to this program as the *software-under-test (SUT)*.

The bug report shows a progressive understanding of the buggy interleaving. The report starts from the stress testing of the code in Figure 1 with 100 threads to reproduce the bug. Then, the programmers collect information from the stack traces of buggy executions to identify problematic code locations and schedules. Finally, they figure out a scenario, given in Figure 2, describing of a suspicious interleaving of three threads A, B, and C executing `testfunc` in Figure 1.

Notice that, this scenario is still partial: It explains the key scheduling decisions to bring the execution to a problematic program state, say **S**, after which it will be highly likely that the bug will manifest. But, the description leaves unspecified the rest of the schedule, from **S** to the assertion violation. By writing a CONCURRIT test, we can specify 1) the known scheduling decisions until **S** and 2) a search for the missing part of the buggy schedule after **S**.

In this section, we demonstrate how to write tests in CONCURRIT to incrementally capture the different stages of understanding of the bug in the bug report. We will start with simple CONCURRIT tests that assume little knowledge about the bug—i.e., having three threads running the code in Figure 1, but not knowing how to interleave them. To begin, we found it useful to first check whether the bug is due to concurrency (Section 2.1), then continue with tests with many possible thread schedules, but that do trigger the bug. These tests will require searching among a large number of thread schedules for a buggy one. We will subsequently incorporate more and more intuition from the bug report to make our test more precise and efficient, by searching among fewer schedules. In this way: (1) we show that a CONCURRIT script can formally and concisely express the intuition of the programmer at various levels of detail,

```
static JSRuntime *rt;
static void* testfunc(void* ignored) {
  JSContext *cx = JS_NewContext(rt, 0x1000);
  if (cx) {
     JS_BeginRequest(cx);
     JS_DestroyContext(cx);
  }
  return NULL;
}
int main(void) {
  rt = JS_NewRuntime(0x100000);
  if (rt == NULL) return 1;
  ... // create threads to run testfunc (and join all)
  return 0;
}
```

Figure 1. Test code manifesting a bug in SpiderMonkey (taken from the related bug report).

Now suppose there are 3 threads, A, B, C running `testfunc`.
Threads A and B call js_DestroyContext and thread˜C calls js_NewContext.
First thread˜A removes its context from the runtime list. That context is not
the last one so thread does not touch rt—>state and eventually calls js_GC.
The latter skips the above check and tries to to take the GC lock.
Before this moment the thread˜B takes the lock, removes its context from the
runtime list, discovers that it is the last, sets rt—>state to LANDING, runs
the-last-context-cleanup, runs the GC and then sets rt—>state to DOWN.
At this stage the thread˜A gets the GC lock, setup itself as the thread that
runs the GC and releases the GC lock to proceed with the GC
when rt—>state is DOWN.
Now the thread˜C enters the picture. It discovers under the GC lock in
js_NewContext that the newly allocated context is the first one. Since
rt—>state is DOWN, it releases the GC lock and starts the first context
initialization procedure. That procedure includes the allocation of the initial
atoms and it will happen when the thread˜A runs the GC.
This may lead precisely to the first stack trace from the comment 4.

Figure 2. Bug scenario, taken from Comment #5 of the bug report, describing an interleaving of threads for the program in Figure 1.

and (2) we highlight the common usage patterns of our DSL that we follow in our experiments. (In Section 5.1 we revisit the general form of these patterns.)

While we demonstrate only a few steps of refining CONCURRIT tests, in reality, we do not always expect to have a quick transition between tests, and there may be more steps before obtaining a practical test. We expect the error information (such as stack traces) from the failing executions in earlier steps will play the role of bug reports to refine tests for the later steps. Although the earlier tests find the bug, our goal is not only to trigger the bug, but to have concise, informative, and targeted (towards the bug) tests.

2.1 First test: Run threads sequentially until completion

To begin, suppose that our intuition about our target interleaving is limited: We know that three distinct threads, A, B, and C, are sufficient to reproduce the bug, but we do not know how to schedule these threads to trigger the error. Indeed, we do not even know whether the bug is sequential or concurrent. Thus, before interleaving the threads, we want to check if the bug is really related to concurrency. For this, we want to run every thread sequentially, i.e., without being interleaved with another thread until its completion. Our first CONCURRIT test is as follows:

```
SearchSeqTest :
1  Tid tA, tB, tC = WaitForDistinctThreads(3, ThreadStarts);
2  while(!HasEnded(tA) || !HasEnded(tB) || !HasEnded(tC)) {
3     Tid t = ChooseThread(tA, tB, tC);
4     RunThreadUntil(t, ThreadEnds);
5  }
```

A CONCURRIT test is run concurrently with the SUT (in Figure 1) and controls the scheduling of the threads in the SUT. Test `SearchSeqTest` demonstrates two kinds of control CONCURRIT can impose on the schedule: (1) It describes how the threads A, B, and C in each execution should be scheduled; in our case, each thread runs sequentially (without interleaving with another thread) until completion. (2) It describes a set of schedules satisfying the constraint in (1) and differ in the scheduling order of threads. In order to realize this control, the SUT is instrumented to interact with the CONCURRIT script. The CONCURRIT script hides the mechanism for this interaction from the programmer and allows her to specify the thread schedule without thinking about the low-level details of how the SUT is controlled.

The diagram in Figure 3 illustrates the interaction between three threads in the SUT and the test `SearchSeqTest`. Function `WaitForDistinctThreads` at Line 1 waits for three distinct

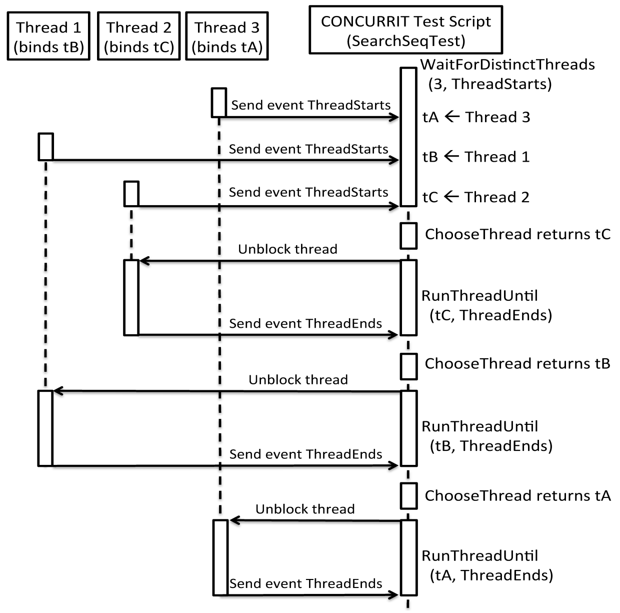

Figure 3. Interaction between the SUT threads and test script.

threads in the SUT to run and reach a point satisfying its second argument (`ThreadStarts`), then binds variables `tA`, `tB`, and `tC` to the identifiers of those threads. The predicate `ThreadStart` indicates that this binding occurs when the threads start executing. On binding, each thread is blocked until re-enabled by the test.

At lines 2-5, the test controls the execution of threads `tA`, `tB`, and `tC` in a loop, which iterates until all the threads terminate (line 2). At each iteration, `ChooseThread` nondeterministically chooses one of the variables `tA`, `tB`, and `tC` (line 3) and assigns it to variable `t`. Then, `RunThreadUntil` enables the chosen thread referred to by `t` to execute (line 4). The second argument of `RunThreadUntil` indicates until when the thread will be allowed to execute. The predicate `ThreadEnds` indicates that thread `t` should keep running until it terminates. While `t` is executing, the other two threads remain blocked. In addition, since the test binds only three threads, all other threads in the SUT remain blocked until the loop ends, after which the control on the SUT imposed by the test is removed.

Controlled nondeterminism. The command `ChooseThread` specifies a nondeterministic choice of given arguments. This nondeterminism is controlled by our test and resolved to implement a search for thread schedules. For example, one can implement random testing by using a random number generator to choose the thread.

In our work, we provide a search strategy that restarts the SUT multiple times and explores a distinct combination of these choices until all distinct combinations are enumerated. This provides the mechanism to search for all possible thread schedules specified by the CONCURRIT test. During this search, for example, if `ChooseThread` chooses an already-terminated thread, `RunThreadUntil` timeouts and restarts the SUT to explore a different schedule. While the choice points for the schedule are indicated explicitly using `ChooseThread`, the underlying search strategy is transparent to the programmer. Thus, as explained in Section 4, one can customize the search independently of the test, for example to follow a depth-first or breath-first order.

When guided by the test `SearchSeqTest`, our search generates 6 thread schedules in which threads `tA`, `tB`, `tC` are run in different orders (but sequentially). However, these schedules do not trigger the assertion violation. This confirms our intuition that the bug is due to concurrency. Thus, we need to allow threads to interleave with each other to trigger the bug.

2.2 Second test: Interleave threads at finest granularity

In our second test we want to interleave the threads. But, suppose we still do not have any knowledge about what interleavings will trigger the bug. Our next step is to allow threads to interleave arbitrarily, and search for the buggy interleaving among these fine-grained interleavings. We can slightly change `SearchSeqTest` to permit such interleaving of threads `tA`, `tB`, and `tC` as follows:

```
SearchAllTest :
1 Tid tA, tB, tC = WaitForDistinctThreads(3, ThreadStarts);
2 while(!HasEnded(tA) || !HasEnded(tB) || !HasEnded(tC)) {
3    Tid t = ChooseThread(tA, tB, tC);
4    RunThreadUntil(t, ReadsMem || WritesMem || CallsFunc|| ThreadEnds);
5 }
```

We modify line 4 to replace the predicate `ThreadEnds` with `ReadsMem || WritesMem || CallsFunc|| ThreadEnds`. The new predicate indicates that thread `t` (chosen at line 3) should run until it attempts to access the memory (read from or write to an address), calls a function, or terminates. When thread `t` attempts to perform one of these operations, the command `RunThreadUntil` ends and leaves thread `t` blocked before that operation. When `RunThreadUntil` is called again with `t` bound to the same thread, it first unblocks the thread to let it complete the operation, and then runs the thread until next operation satisfying the predicate.

As it allows threads to interleave at fine granularity, our second test `SearchAllTest` can generate the buggy thread schedule violating the assertion. However, the buggy schedule is generated after exploring *millions* of schedules of the test. This means, whenever we want to reproduce the bug using this test, we have to wait for *days* to see the assertion violation. Although the test allows us to examine all schedules, there are a huge number of them. Moreover, the bug is triggered by a small number of tricky interleavings; thus, most of the schedules do not create the buggy situation. Thus, it will not be practical to use `SearchAllTest` for the purpose of reproducing the bug. (It is for this reason that traditional model checking is often impractical to use for large software.)

2.3 Third test: Localize search to suspicious states

The real power of CONCURRIT can be seen when we need to restrict the search space of interleavings. Note that, `SearchAllTest` runs the search for the entire lifetime of the threads. After examining the executions with assertion violations generated by `SearchAllTest`, we next localize the search for interleavings to a shorter period of the execution in order to enumerate fewer interleavings and to more quickly reach the intended state at which the assertion is violated. We refine our test as follows:

```
SearchInBuggyTest :
1 Tid tA, tB, tC = WaitForDistinctThreads(3, EntersFunc(JS_NewContext));
2 RunThreadsUntil(tA, tB, EntersFunc(JS_DestroyContext));
3 RunThreadUntil(tA, InFunc(js_GC) && ReadsMem(&rt-> state));
4 RunThreadUntil(tB, ThreadEnds); // thread B does not involve after this point
5 RunThreadUntil(tA, InFunc(js_GC) && WritesMem(&rt-> gcThread));
6 while(InFunc(tA, JS_DestroyContext) || InFunc(tC, JS_NewContext)) {
7    Tid t = ChooseThread(tA, tC);
8    RunThreadUntil(t, ReadsMem || WritesMem || CallsFunc|| ThreadEnds);
9 }
```

We first notice that the problematic portion of the interleaving is after threads A and B call `JS_DestroyContext` and thread C calls `JS_NewContext`. Thus, we want to only examine interleavings starting not from the beginning of the execution, but after the threads enter these functions. We incorporate this new insight to our CONCURRIT test as follows. First, we replace the condition `ThreadStart` at line 1 with `EntersFunc(JS_NewContext)`. The new version of `WaitForDistinctThreads` waits until the threads all start executing `JS_NewContext`—and then get blocked. Then, the command `RunThreadsUntil` at line 2 runs threads `tA` and `tB` (permitting them to interleave) until both threads start running `JS_DestroyContext`, at which point they are blocked.

Furthermore, at lines 3-5 we explicitly schedule the threads to lead the execution towards a problematic state (discussed below). The search for distinct interleavings starts after this point and is performed at lines 6-9. The body of the loop to interleave threads differs from that of `SearchAllTest` in two ways. First, as thread B terminates (at line 4) before the search starts, we remove references to `tB` from the loop. Second, the loop condition indicates that we are only interested in distinct interleavings of threads while running `JS_NewContext` and `JS_DestroyContext`.

Compared to `SearchAllTest`, our test `SearchInBuggyTest` is more effective in reproducing the bug: it generates the buggy interleaving after exploring less than 10 schedules of the test.

Capturing the programmer's knowledge in the bug report. Test `SearchInBuggyTest` indeed concisely and formally expresses the interleaving scenario in Figure 2. Our test also includes a search for the final portion of the interleaving to address the (missing) rest of the scenario. While the test scenario in Figure 2 explains our intuition about why the assertion is violated, the test formally documents and implements this intuition to reproduce the bug.

In detail, we realize that the assertion is violated because in a buggy execution there exists a problematic state **S** at which thread A starts a garbage collection routine `js_GC` to deallocate some shared data structures of the JavaScript engine, and thread C starts initializing these data structures. The scenario describes an interleaving to lead the execution towards state **S** and the commands between lines 1-5 of `SearchInBuggyTest` guides the threads to follow this interleaving. During this interleaving thread B terminates (at line 4), and the rest of the execution after **S** involves only threads A and C executing functions `JS_DestroyContext` and `JS_NewContext`, respectively. Interleaving of A and C after **S** results in a data race between A and C on a shared data structure, leading to an assertion violation about the data structure. Note that, our knowledge about the buggy interleaving is still partial, because we do not know how A and C should interleave after **S** to trigger the assertion violation. Thus, at lines 6-9, we search for interleavings A and C during this final part of the execution.

Uncontrolled nondeterminism. In addition to the controlled nondeterminism in command `ChooseThread`, `SearchInBuggyTest` also allows uncontrolled nondeterminism. Except when specifically blocked and sequentialized by the CONCURRIT script, the threads in the SUT are allowed to run in parallel following a nondeterministic schedule. For example, at Lines 1 and 2, the threads run in parallel until they start executing functions from `JS_NewContext` and `JS_DestroyContext`. The test does not control the nondeterminism in the thread schedule until they start executing these functions. Thus, a search guided by `SearchInBuggyTest` may generate another execution in which the threads follow a different interleaving before they enter these functions; such interleavings are not distinguished from each other. Moreover, after Line 1, any threads other than `tA`, `tB`, and `tC` run in parallel without any interruption by the CONCURRIT test. By embracing uncontrolled nondeterminism in this way, a CONCURRIT test can tolerate changes in the execution context across different executions of the SUT and can still explore different schedules. In addition, our tests highlight that the CONCURRIT script can express a very complicated schedule in a compact form, as it specifies only the key scheduling decisions and choices. Therefore, a programmer can conveniently use CONCURRIT to test her partial knowledge about the thread schedules without needing to specify uninteresting and irrelevant parts of the schedule.

2.4 Final test: Generate exact buggy interleaving

In `SearchInBuggyTest` we have localized the search (lines 6-9) for the latter, short part of the schedule. As a result, each interleaving explored by the loop in `SearchInBuggyTest` contains few context switches of threads `tA` and `tC`. Moreover, since we have also increased the likelihood of hitting the bug in `SearchInBuggyTest`, we will have many buggy interleavings to compare and find the common scheduling decisions in the latter part of the schedule. Thus, it becomes manageable to analyze the latter part of the interleavings and extract a test without search that will trigger the bug at every execution.

To develop the final, exact schedule, we select buggy executions with the least number of context switches and (manually) alternated two refinement steps: *Strengthen:* Using the execution trace, translate an exact, buggy schedule to a CONCURRIT test, i.e., express a sequence of operations by the same threads to a `RunThreadUntil` command. *Weaken:* Try to remove some `RunThreadUntil` commands and check if the bug is still triggered. After several refinement steps, we figure out an exact interleaving after the problematic state **S** to reach the assertion violation, and replace the loop in `SearchInBuggyTest` with a sequence of `RunUntilThread` commands to guide the execution of the SUT along the buggy interleaving. Our final test `ExactScheduleTest` is as follows:

```
ExactScheduleTest :
Tid tA, tB, tC = WaitForDistinctThreads(3, EntersFunc(JS_NewContext));
RunThreadsUntil(tA, tB, EntersFunc(JS_DestroyContext));
RunThreadUntil(tA, InFunc(js_GC) && ReadsMem(&rt−> state));
RunThreadUntil(tB, ThreadEnds);
RunThreadUntil(tA, InFunc(js_GC) && WritesMem(&rt−> gcNumber));
RunThreadUntil(tC, EntersFunc(js_AddRoot));
RunThreadUntil(tA, ReturnsFunc(js_GC)); // violates assertion!
```

Our final test concisely describes few key scheduling decisions causing the bug and leaves other parts of the thread schedule unspecified. The control imposed by the CONCURRIT script is enough to guide the SUT to the assertion violation at every execution. Thus, we can use this test to file a bug report and to check if any fix for the bug prevents the problem; after the fix, we can add the test to our regression suite.

In summary, we demonstrated an iterative process for one to refine a CONCURRIT test by incorporating new insights about the target schedules. At each step, we fixed a longer, former portion of the target interleaving and localized the search for unknown portion of the interleaving around a smaller region of the interleaving space. Finally, we were able to simplify our test to describe an exact, buggy schedule in a concise and formal form.

3. DSL for controlling thread schedules

In this section, we provide a low-level, imperative language to describe and control intended thread schedules of the SUT.

3.1 Software-under-test (SUT)

We do not assume any specific syntax and semantics for the Software-under-Test (SUT)—we leave abstract (1) the structure of SUT-states and (2) transitions between such states. Let $SUTState$ refer to the set of all SUT-states, S range over $SUTState$, and $S_0 \in SUTState$ be the unique initial state of SUT.

We assume that each SUT-state consists of a set of threads, each of which is referred to via a unique thread identifier. Let Tid be the set of all thread identifiers and t range over Tid. In case the SUT consists of multiple processes, the set Tid contains all the threads contained in these processes.

To present the operation of a CONCURRIT test, we assume a standard interleaving semantics for the execution of threads in the SUT: At any time only one thread is allowed to execute, modifying its own local variables, and possibly, some global variables. We also abstract away the memory model issues by assuming a sequentially consistent memory model. We use the notation $S \xrightarrow{t} S'$ to indicate that a thread $t \in Tid$ in the SUT can execute from state S and leads to SUT-state S'.

In addition, the SUT code is instrumented, so that when a thread t visits an instrumentation point, it generates an event carrying information about the operation to be performed next by t and blocks. (See Section 6 for details of this instrumentation). For this, we extend the notation for SUT-transitions as follows. We write $S \xrightarrow{t} S'/e$ to indicate that thread t also generates an event e as a result of the state transition.

Event. We represent each *event* as a record from fields to values describing an operation performed by a thread, such as accessing a memory location, entering or exiting a function, and terminating. Let $Event$ be the set of events, and e range over $Event$.

Given an event e, we write $e.f$ to refer to field f of the event. Each event contains at least two fields, $kind$ and tid, to indicate the type of the operation for which the event is generated and the generating thread's identifier, respectively. Other fields are used to store more information about the operation, for example, $addr$ field for the memory address read or written, $func$ field for the function being called, and pc field for the related source location.

CONCURRIT scripts use a special form of boolean expression, called *event predicates*, to represent sets of events. Let `EPred` contain all event predicates and `p` range over `EPred`. Following is the list of event predicates we commonly use in our tests:

`WritesMem`	$\equiv$	λe. e.kind == MemWrite
`WritesMem(a)`	$\equiv$	λe. e.kind == MemWrite && e.addr == a
`EntersFunc(f)`	$\equiv$	λe. e.kind == FuncEnter && e.func == f
`CallsFunc`	$\equiv$	λe. e.kind == FuncCall
`ThreadEnds`	$\equiv$	λe. e.kind == ThreadEnd
`AtControl(x)`	$\equiv$	λe. e.kind == Control && e.pc == x

Predicate `AtControl` is used to refer to events generated at particular source locations (control point) in the SUT code.

The events generated by SUT threads are communicated to the CONCURRIT script, and they provide the mechanism to interact with and control the execution of the SUT. In particular, generating an event causes the originating thread to block until the CONCURRIT script observes the event and explicitly enables the generating thread to continue.

3.2 CONCURRIT: Syntax

Figure 4 shows the core syntax of CONCURRIT. We will extend this syntax in Section 4 to support exploring multiple test executions.

A statement in our DSL is denoted by `C`. Our DSL extends an imperative language with two special commands: `select` and `release`. (Command `endrun` represents an instruction that marks the end of the execution.) These special commands can be implemented as a library of procedures on top of an imperative language, in our case, C/C++. Thus, we follow the C language syntax for standard imperative constructs and expressions in CONCURRIT.

Symbols `b`, `t`, and `e` refer to CONCURRIT variable names storing Boolean values, thread identifiers, and events, respectively. The variables have types `bool`, `Tid`, and `Event`, respectively. We also use `x` to refer to a variable of an arbitrary type.

SUT-STEP-WITHOUT-EVENT
$$\frac{t \in Tid \qquad E(t) = \bot \qquad S \xrightarrow{t} S'}{(S, D, E, \mathtt{C}) \dashrightarrow (S', D, E, \mathtt{C})}$$

SUT-STEP-WITH-EVENT
$$\frac{t \in Tid \qquad E(t) = \bot \qquad S \xrightarrow{t} S'/e}{(S, D, E, \mathtt{C}) \dashrightarrow (S', D, E[t \mapsto e], \mathtt{C})}$$

SELECT
$$\frac{t \in D \Downarrow \mathtt{T} \qquad E(t) = e \qquad D' = D[\mathtt{e} \mapsto e]}{(S, D, E, \mathtt{e} = \mathtt{select(T)};\mathtt{C}) \dashrightarrow (S, D', E, \mathtt{C})}$$

SELECT-TIMEOUT
$$\frac{\begin{array}{c}\forall t \in D \Downarrow \mathtt{T}.\ E(t) = \bot \\ (\textit{Time limit for } \mathtt{select} \textit{ expires})\end{array}}{(S, D, E, \mathtt{e} = \mathtt{select(T)};\mathtt{C}) \dashrightarrow (S, D, E, \mathtt{endrun})}$$

RELEASE
$$\frac{\begin{array}{c}\forall t \in D \Downarrow \mathtt{T}.\ (E(t) \neq \bot\ \wedge E'(t) = \bot) \\ \forall t \notin D \Downarrow \mathtt{T}.\ E'(t) = E(t)\end{array}}{(S, D, E, \mathtt{release(T)};\mathtt{C}) \dashrightarrow (S, D, E', \mathtt{C})}$$

IF
$$\frac{\mathtt{C}' = (D(\mathtt{b}) = \textit{true})\ ?\ (\mathtt{C}_1;\mathtt{C})\ :\ (\mathtt{C}_2;\mathtt{C})}{(S, D, E, \mathtt{if(b)}\ \{\mathtt{C}_1\}\ \mathtt{else}\ \{\mathtt{C}_2\};\mathtt{C}) \dashrightarrow (S, D, E, \mathtt{C}')}$$

WHILE
$$\frac{\mathtt{C}' = (D(\mathtt{b}) = \textit{true})\ ?\ (\mathtt{C}_1;\mathtt{while(b)}\ \{\mathtt{C}_1\};\mathtt{C})\ :\ \mathtt{C}}{(S, D, E, \mathtt{while(b)}\ \{\mathtt{C}_1\};\mathtt{C}) \dashrightarrow (S', D, E, \mathtt{C}')}$$

Figure 5. The operational semantics of CONCURRIT.

$\mathtt{b}, \mathtt{t}, \mathtt{e}, \mathtt{x} \in Var$

T	::=	AnyThread | t | T+t | T−t	*Thread expressions*
C	::=	e = select(T)	*Wait for event from thread(s)*
	|	release(T)	*Unblock thread(s)*
	|	endrun	*End of execution*
	|	if(b) C else C	*Conditional*
	|	while(b) C | break	*Loop*
	|	C; C	*Sequential composition*
	|	...	*Other imperative constructs*

Figure 4. Syntax of CONCURRIT.

3.3 CONCURRIT: Semantics

We next give an operational semantics to CONCURRIT. For this, we think of the software-under-test (SUT) and the test written in CONCURRIT running together as a single system, in which the SUT and the DSL script run concurrently and in interaction with each other (events provide the mechanism for their interaction). Thus, we define a *test state* as the composed state of the SUT and the DSL script in CONCURRIT.

3.3.1 Test state

A test state is described by a tuple $(S, D, E, \mathtt{C})$:

- $S \in SUTState$ is a state of SUT as described in Section 3.1. Note that we abstract out the details of SUT states.
- $D \in DSLState$ is the state of the DSL script written in CONCURRIT. Let Var be a set of variable names. State D is a map from Var to $Value$. Let $dom(D)$ denote the set of variables defined at state D. We write $D(\mathtt{x})$ to denote the value of a variable $\mathtt{x} \in dom(D)$ in state D. We write $D[\mathtt{x} \mapsto v]$ to denote the DSL state that agrees with D for all variables except that $\mathtt{x}$ is mapped to value v.
- E is a map from Tid to $Event$. For a thread $t \in Tid$, $E(t) = \bot$ indicates that t has no event associated with it, and $E(t) \in Event$ indicates that t is blocked at an event and waiting for the CONCURRIT script to unblock it. We write $E = [\lambda t \in Tid.\ \bot]$ to indicate that no thread in E is mapped to an event. We use similar notation to D to update the map E (e.g., $E[t \mapsto e]$).
- $\mathtt{C}$ is the rest of the DSL program (a statement from Figure 4) to evaluate next.

3.3.2 Test executions

Figure 5 gives an operational semantics for an execution of the SUT composed by the test in CONCURRIT. Such an execution is expressed as a sequence of test-state transitions governed by the rules in Figure 5:

$$(S_0, D_0, [\lambda t \in Tid.\ \bot], \mathtt{C}_0; \mathtt{endrun}) \dashrightarrow^* (S_n, D_n, E_n, \mathtt{endrun})$$

In the initial test-state, S_0 and D_0 denote the unique initial states of the SUT and the DSL, and $\mathtt{C}_0$ denotes the CONCURRIT script written by the programmer. Notice that, we sequentially compose the DSL statement $\mathtt{C}_0$ with special command endrun. During the execution, the statement $\mathtt{C}_0$; endrun is modified by the operational semantics rules to indicate the next DSL statement to be evaluated, and the execution terminates when the DSL statement is fully evaluated to endrun.

Without loss of generality, we use an interleaved semantics for the execution, where at any time either an SUT thread or the DSL script may take a transition.

SUT-transitions. As stated by rules SUT-STEP-WITHOUT-EVENT and SUT-STEP-WITH-EVENT, a thread t in the SUT may execute and lead the SUT from state S to a new state S' (expressed by $S \xrightarrow{t} S'$). The transitions by an SUT thread t is enabled only when $E(t) = \bot$ holds, i.e., there is no event associated with t in the current test state. If thread t generates an event e as a result of this transition, it is associated with that event using the map E (see SUT-STEP-WITH-EVENT). This causes thread t to be blocked until the DSL script executes a release command on thread t to remove this mapping (see rule RELEASE).

DSL-transitions. Each DSL transition evaluates the current script $\mathtt{C}$ to result in a new script $\mathtt{C}'$, and possibly modifies the internal DSL state D and the mapping E. These transitions are governed by the semantics rules other than SUT-STEP-WITHOUT-EVENT and SUT-STEP-WITH-EVENT. Transitions by the DSL do not affect the SUT-state S. The DSL script can only control the execution of the SUT by modifying the map E, which affects the further transitions of the SUT by, for example, unblocking a thread waiting on an event.

3.3.3 Controlling thread schedule with select and release

Thread expressions. The DSL commands select and release operate on a set of thread identifiers. We use thread expressions (T of type TExpr) to identify a set of thread identifiers. Syntax for thread expressions is given in Figure 4. First, the special DSL constant AnyThread represents the set of all thread identifiers, i.e., Tid. Second, each thread variable t (storing the identifier of a thread) is also treated a thread expression representing the set $\{D(\mathtt{t})\}$. Finally, we overload the + and − operators to include or exclude a thread identifier to/from the set.

We write $D \Downarrow \mathtt{T}$ to refer to the set of thread identifiers (subset of Tid) represented by the expression T in D. This set is defined recursively as follows:

$$\begin{array}{llll} D \Downarrow \mathtt{AnyThread} = Tid & & D \Downarrow (\mathtt{T} + \mathtt{t}) = D \Downarrow \mathtt{T} \cup \{D(\mathtt{t})\} \\ D \Downarrow \mathtt{t} & = \{D(\mathtt{t})\} & D \Downarrow (\mathtt{T} - \mathtt{t}) = D \Downarrow \mathtt{T} \setminus \{D(\mathtt{t})\} \end{array}$$

We say that a thread with identifier t *satisfies* `T` (or `T` is satisfied by thread t) at a state D if $t \in D \Downarrow \mathtt{T}$.

Waiting for an event. The command $\mathtt{e} = \mathtt{select}(\mathtt{T})$ takes a thread expression and waits for a thread whose identifier is in the set $D \Downarrow \mathtt{T}$ to generate an event. The command then binds variable `e` to the generated event. For example, $\mathtt{e} = \mathtt{select}(\mathtt{AnyThread})$ waits for any thread, $\mathtt{e} = \mathtt{select}(\mathtt{t})$ waits for thread `t`, and $\mathtt{e} = \mathtt{select}(\mathtt{AnyThread} - \mathtt{t})$ waits for any thread except `t` to generate an event.

Semantics rule SELECT governs this operation of `select`. Note that an event generated by a thread t is recorded at $E.t$, so the rule simply queries E to retrieve this event.

Timeout. Due to an incorrect test script or uncontrolled nondeterminism in the SUT (as explained in Section 2), no thread in $D \Downarrow \mathtt{T}$ may generate an event from the current test state. In order to detect such situations, we allow the implementation of a timeout mechanisms. For this, we include the rule SELECT-TIMEOUT, which allows `select` to end the execution by evaluating the current DSL statement to `endrun`, when no thread in $D \Downarrow \mathtt{T}$ is associated with an event.

***Nondeterminism in* `select`.** Notice that rules SELECT and SELECT-TIMEOUT bring some nondeterminism to the semantics. For example, consider a command $\mathtt{e} = \mathtt{select}(\mathtt{T})$. First, the command can choose an arbitrary thread from the set $D \Downarrow \mathtt{T}$, or, before taking effect it can even wait for a state in which more threads generate events and satisfy `T`. Second, if no thread in $D \Downarrow \mathtt{T}$ has generated an event, the command may evaluate to `endrun` before waiting a thread to generate an event. For the former case, our high-level constructs in Section 5 give examples to controlling this nondeterminism. For the latter case, the programmer can set, in the CONCURRIT test, a time limit for each invocation of a `select` command.

Unblocking threads. The command `release(T)` takes a thread expression and unblocks all threads whose identifiers are in the set $D \Downarrow \mathtt{T}$. This is governed by the rule RELEASE. The rule requires that all threads that are specified by `T` are associated with an event. Thus, `release` should be called for threads that have already generated and associated with an event at the current test state. A common way to ensure this condition, which we consistently followed in our case studies, is to match each `release` on a thread t to a previous `select` on the same thread t.

4. Searching for thread schedules

Having introduced the CONCURRIT constructs to control the scheduling of threads, we next extend our DSL to express a set of schedules and present mechanisms to search for these schedules in the presence of uncontrolled nondeterminism.

To accommodate the notion of search in CONCURRIT, we first add to the syntax of CONCURRIT a new command: (Let k refer to a constant value.)

C	::=	$\cdots$	*Constructs from Figure 4*
	|	$\mathtt{x} = \mathtt{choose}(\mathtt{k_1}, \ldots, \mathtt{k_n})$	*Ask oracle to choose a value*

The new construct `choose` is used to guide the execution (in addition to `select` and `release`) to support the search for schedules. To explain the semantics of `choose`, we include in our formalism an *oracle* denoted, $\mathcal{O}$, which resolves controlled nondeterminism in the test. We extend the test state (defined in Section 3.3.1) to include an oracle, and add to the operational semantics two new rules shown in Figure 6. The original rules in Figure 4 remain the same, except that they preserve $\mathcal{O}$ as is during the transition. The oracle $\mathcal{O}$ may change only by the new rules in Figure 6.

CHOOSE

$$\frac{\mathcal{O}.choose(\mathtt{k_1}, \ldots, \mathtt{k_n}) = (k, \mathcal{O}') \qquad k \in \{\mathtt{k_1}, \ldots, \mathtt{k_n}\} \qquad D' = D[\mathtt{x} \mapsto k]}{(S, D, E, \mathtt{x} = \mathtt{choose}(\mathtt{k_1}, \ldots, \mathtt{k_n}); \mathtt{C}, \mathcal{O}) \dashrightarrow (S, D', E, \mathtt{C}, \mathcal{O}')}$$

RESTART

$$\frac{\mathcal{O}.restart() = Some(\mathcal{O}')}{(S, D, E, \mathtt{endrun}, \mathcal{O}) \dashrightarrow (S_0, D_0, [\lambda t \in Tid.\ \bot], \mathtt{C_0}; \mathtt{endrun}, \mathcal{O}')}$$

Figure 6. The extended operational semantics of CONCURRIT for search. We keep the original rules in Figure 5 except that we include an oracle $\mathcal{O}$ to the test state and preserve $\mathcal{O}$ during the transition.

Choose. Rule CHOOSE governs the operation of the command $\mathtt{x} = \mathtt{choose}(\mathtt{k_1}, \ldots, \mathtt{k_n})$. The command is used to request an *input* from the oracle $\mathcal{O}$ to be used to guide the current execution. Operation $\mathcal{O}.choose(k_1, \ldots, k_n)$ returns a pair $(k, \mathcal{O}')$, where k is the value chosen by the oracle from $k_1, \ldots, k_n$, and $\mathcal{O}'$ is the new state of the oracle. The oracle can choose nondeterministically or it can track choices across multiple executions to implement a search.

Restart. Rule RESTART allows the oracle $\mathcal{O}$ to start a new execution of the SUT when the current one terminates (when the CONCURRIT script evaluates to `endrun`). For this, we define the $\mathcal{O}.restart()$ operation to return either $Some(\mathcal{O}')$ or $None$. If the oracle decides to start a new execution, it returns $Some(\mathcal{O}')$, where $\mathcal{O}'$ is the oracle to guide the SUT during the new execution. In this case, the SUT and the CONCURRIT script are restarted from their initial states. Recall that, in the initial test-state, the CONCURRIT script $\mathtt{C_0}$ is sequentially composed with `endrun`. If the oracle decides that there are no more executions to explore, it terminates the overall test by returning $None$.

Modularity of Test Scripts and Oracles. The interface of the oracle cleanly separates (1) the description of a set of thread schedules in CONCURRIT and (2) the exploration of these schedules. The oracle guides the exploration in (2) by defining a search strategy and related heuristics transparently to the CONCURRIT script. To explain this separation, consider the following test:

```
InterleaveTwoThreads :
1  Event e1 = select(AnyThread); Tid t1 = e1.tid;
2  Event e2 = select(AnyThread − t1).tid; Tid t2 = e2.tid;
3  while(!HasEnded(t1) || !HasEnded(t2)) {
4     Tid t = choose(t1, t2);      // Oracle chooses one of the threads.
5     do {                         // Loop until thread t accesses the memory.
6        release(t);               // Enable thread t to execute.
7        Event e = select(t);      // Wait until thread t blocks at another event.
8     } while(!ReadsMem(e) && !WritesMem(e));
9  }
```

The test binds variables `t1` and `t2` to two distinct threads at Line 1 and 2, respectively. At each outer loop iteration, one of the threads `t1` or `t2` is chosen by the `choose(t1, t2)` construct at Line 4. Then, the chosen thread `t` is enabled to execute using `release` at Line 6, until the thread is blocked at another event. When the chosen thread is blocked at an event before an access (read or write) to the memory, the test repeats the outer loop and choses a new thread. Irrespective of the choice made by the oracle $\mathcal{O}$ at Line 4, the test describes a set of interleavings, say $\mathbf{I}$, of two distinct threads where a new scheduling decision is made (at Line 4) after every memory access event. The described interleaving ends when both threads terminate (`HasEnded(t)` at Line 3 returns true iff thread `t` has terminated). Different oracles, however, will differ in the order in which the interleavings are explored.

Implementing oracles. With our oracle interface—`choose` and `restart`—it is straightforward to implement various search strate-

```
$\mathcal{O}.choose(k_1, \ldots, k_n)$:
  if $\mathcal{O}.index > |\mathcal{O}.\tau|$ then
    // EXTEND PREFIX: Record chosen value in the prefix
    let $K = \{k_1, \ldots, k_n\}$
    $k$ = nondeterministically choose from set $K$
    $\mathcal{O}.\tau.append((k, K \setminus \{k\}))$
  else
    // REPLAY PREFIX: Use previously chosen value in the prefix
    let $(k, K) = \mathcal{O}.prefix[\mathcal{O}.index]$
  endif
  $\mathcal{O}.index = \mathcal{O}.index + 1$
  return $(k, \mathcal{O})$

$\mathcal{O}.restart()$:
  $R = \{i \mid 1 \leqslant i \leqslant |\mathcal{O}.\tau| \wedge \mathcal{O}.\tau[i] = (k, K) \wedge K \neq \varnothing\}$
  if $R \neq \varnothing$ then
    $m = max(R)$
    let $(k, K) = \mathcal{O}.\tau[m]$ and $k' \in K$
    $\mathcal{O}.\tau.pruneAfter(m - 1)$
    $\mathcal{O}.\tau.append((k', K \setminus \{k'\}))$
    $\mathcal{O}.index = 1$
    return $Some(\mathcal{O})$
  else
    return $None$
  endif
```

Figure 7. Oracle implementing depth-first search.

gies, such as breadth-first and depth-first search. In the following, we give an abstract, declarative description of an oracle implementing search. To make our point concrete, we explain how to instantiate this oracle for depth-first search; other search techniques and heuristics (e.g., [17]) can be integrated to our framework following a similar methodology.

4.1 Implementing oracles for search

Our oracle $\mathcal{O}$ tracks a *trace* of each execution, denoted τ. The trace consists of the inputs and outputs interchanged between the oracle and the CONCURRIT test—in particular the sequence of choices taken by the oracle in response to `choose` commands. The oracle also keeps a set $\mathcal{T}$ of traces for the complete executions it observes.

Initially, set $\mathcal{T}$ is empty, and the oracle starts with an empty trace τ. During the execution, calls to operation $\mathcal{O}.choose()$ extend trace τ. Note that, when the execution terminates, the semantic rule RESTART invokes $\mathcal{O}.restart()$ to decide whether to start a new execution or to terminate the search. Operation $\mathcal{O}.restart()$ first adds τ to set $\mathcal{T}$, and then computes from $\mathcal{T}$ a new trace τ' satisfying the following: *(i) Trace τ' is non-empty, (ii) it is not a prefix of another trace in $\mathcal{T}$, and (iii) the longest prefix τ'' of τ' such that $\tau'' \neq \tau'$ is a prefix of some trace in $\mathcal{T}$.* Thus, trace τ' may be used to generate an execution (particularly, a thread schedule) distinct from the previously explored ones. If no such τ' exists, $\mathcal{O}.restart()$ returns $None$, which terminates the exploration. Otherwise, during the next execution, the oracle first replays τ'—i.e., it uses the records in τ' to respond to calls to $\mathcal{O}.choose()$. When all records in τ' are used, the oracle extends τ' to a complete trace in the rest of the execution.

4.1.1 Oracle for depth-first search

In our framework, we implement search by following a depth-first search (DFS) strategy. Figure 7 gives an implementation of the oracle $\mathcal{O}$ following a depth-first search strategy. We implement the trace τ as a sequence of pairs (k, K). Each pair records in the first element k a choice taken by the oracle in response to a `choose` command. The second element K is the set of values that has not been chosen yet. The oracle also keeps an index $index$, which points to an element of τ to be considered next while replaying the trace.

```
1  Tid WaitForThread
       (TExpr T, EPred p) {
2    while(true) {
3      Event e = select(T);
4      if(p(e)) return e.tid;
5      release(e.tid);
6    }
7  }

8  TidList WaitForDistinctThreads
       (int n, EPred p) {
9    TExpr T = AnyThread;
10   TidList ts;
11   while(0 < n--) {
12     Tid t = WaitForThread(T, p);
13     ts.add(t);
14     T = T − t;
15   }
16   return ts;
17 }

18 void RunThreadsUntil
       (Tid t1, ..., tn, EPred p) {
19   TExpr T = t1 + · · · + tn;
20   release(T);
21   int c = n;
22   while(0 < c−−) {
23     Tid t = WaitForThread(T, p);
24     T = T − t;
25   }
26 }

27 Tid ChooseThread
       (Tid t1, ..., tn) {
28   int k = choose(1, ..., n);
29   switch(k) {
30     case 1 : return t1;
31     ...
32     case n : return tn;
33   }
34 }
```

Figure 8. Our test library with high-level functions

For the entire search, the oracle modifies the trace τ as a DFS stack, which eliminates the need for additional traces. Thus, during the $\mathcal{O}.restart()$ operation, the set $\mathcal{T}$ only contains the last explored trace τ. The oracle computes the new trace τ' (to replay in the next execution) from τ by backtracking to the last record (k, K) of a `choose` command in the trace such that K contains at least one unchosen element, say k', and updating that record to $(k', K \setminus \{k'\})$. This ensures that during the replay of τ' the oracle responds to the corresponding invocation of `choose` by returning k'.

5. A high-level library for writing concise tests

While the core CONCURRIT DSL introduced in Sections 3 and 4 is useful to formally explain the control obtained by a CONCURRIT test, the low-level primitives in the DSL may not be appropriate for programmers to write concise tests. In this section, we show how these low-level primitives can be combined to implement higher-level constructs, as a library of functions. Our library is shown in Figure 8. We found this library sufficient and convenient to write tests for our benchmarks (Section 7). In fact, we conclude this section by describing common usage patterns of the library that we have learned from our experiments.

Waiting for a thread. We define two functions to wait for threads. Function `WaitForThread` takes a thread expression `T` and an event predicate `p`, and repeatedly runs the threads satisfying `T` until one thread in `T` generates an event satisfying `p`. A common use of `WaitForThread` is with `T = AnyThread`, to enable all threads in the SUT until the expected event is generated.

Function `WaitForDistinctThreads` waits for a fixed number of threads each to generate an event satisfying `p`. Once a thread generates such an event, we ensure at Line 14 that the thread is no longer run by removing its identifier from the thread expression `T`.

Running threads. The library functions `WaitForThread` and `WaitForDistinctThreads` return one or a set of blocked thread identifiers, respectively. Function `RunThreadsUntil` takes a list of already-blocked threads and reenables them (via `release`). Then, the function runs each reenabled thread until the thread generates an event satisfying predicate `p`. For readability, when using only a single thread identifier `t`, we call the function `RunThreadUntil(t, epred)`.

Selecting a thread variable. We define function `ChooseThread` for selecting from a given set of thread variables, using the primitive `choose` explained in Section 4. Thus, we use this function when we want to introduce a scheduling point for which all possible choices must be explored.

5.1 Test patterns using high-level constructs

In this section, we present common test patterns. Most of our patterns are based on the following generic pattern:

```
GenericPattern(int n, EPred p_start, p_schedule, BExpr loop_cond) :
1  Tid t1, ..., tn = WaitForDistinctThreads(n, p_start);
2  ...... // Run threads t1, ..., tn to follow a fixed interleaving
3  while(loop_cond) {
4     Tid t = ChooseThread(t1, ..., tn);
5     RunThreadUntil(t, p_schedule);
6  }
```

The pattern has four parameters: the number (n) of threads to be controlled during the test, the event predicates, p_start and p_schedule, and a boolean expression loop_cond. We first wait for n threads until each of the n threads generates an event satisfying p_start (line 1). Then, we use functions of our library in Figure 8 to guide the threads to follow a particular schedule (line 2). Finally, we schedule threads in a loop–at each iteration running one of the threads until it generates an event satisfying p_schedule. Then, the SUT is restarted to explore a new thread interleaving.

SearchAll: Traditional model checking. In order to apply model checking to the SUT in the traditional way, we instrument the entire code for the SUT. In particular, the instrumentation ensures that (i) every memory access (read and write) and (ii) every operation that may cause the executing thread to block (e.g., synchronization operations such as acquiring a mutex) generate an event. We over-approximate (ii) by inserting instrumentation before every function call. Under the assumption that the only source of non-determinism in the program is the thread schedule, rescheduling threads at (1) and (2) is sufficient to enumerate all possible thread schedules of the SUT. The pattern SearchAll instantiates our generic pattern GenericPattern as follows: (1) p_start = ThreadStart. (2) Line 2 is omitted. (3) p_schedule = (ReadsMem ‖ WritesMem ‖ CallsFunc ‖ThreadEnds). (4) loop_cond =!HasEnded(t1) || ⋯ || !HasEnded(tn).

SearchInFunc: Restricting search to particular functions. The pattern SearchInFunc restricts the search for the interleaving of threads within a set of functions, say **F**. In this pattern, we modify GenericPattern as follows: (1) We only instrument the code so that a thread generates events only when executing a function in **F**. (2) p_start = EntersFunc(f1) ‖ ⋯ ‖ EntersFunc(fm), where **F** = f1, ..., fm. (3) Similarly to SearchAll, Line 2 is omitted. (4) loop_cond = InFunc(f1) || ⋯ || InFunc(fm), which holds if at least one of t1, ..., tn is running a function from **F**.

SearchInFuncLS: Interleaving threads at large steps. For some benchmarks, we found it useful to limit the scheduling points to a few special locations in the SUT code. For this, we manually instrument these locations in the SUT code to generate user-defined events of the form AtControl(pc). In particular, we associate each of these locations with a user-defined program counter (pc), an integer value. Then, we extend the pattern SearchInFunc by setting p_schedule to (AtControl ‖ ThreadEnds).

SearchInBuggy: Starting search from a suspicious state. Similarly to the pattern SearchInFunc, SearchInBuggy searches for interleavings within a set **F** of functions. However, instead of starting the search from the beginning of the functions, this pattern includes a fixed schedule before starting the search (at line 2 of GenericPattern). This fixed schedule is benchmark-specific and is described by few lines each running RunThread(s)Until.

ExactSchedule. This pattern describes the exact thread schedule intended by the programmer to generate, for example, to document a buggy interleaving in a reliable way. In this case, the schedule only consists of line 1 of GenericPattern and a sequence of RunThread(s)Until, omitting the search loop at lines 3-8.

SearchInFuncCB: Context-bounded search of interleavings. There are many reduction techniques for software model checking, including partial-order reduction [5, 7], preemption bounding [19], and fair stateless model checking [20]. These optimizations are orthogonal to our technique. One can implement these optimizations within the test script instead of modifying the underlying search algorithm (i.e., the interface for oracle $\mathcal{O}$). In our experiments, we found context bounding quite useful to reduce the interleavings searched when using SearchInFunc. The following is an implementation of the context-bounded version of GenericSearch.

```
GenericPatternCB(int n, EPred p_start, p_schedule,
                 BExpr loop_cond, int context_bound) :
1  Tid t1, ..., tn = WaitForDistinctThreads(n, p_start);
2  ...... // Run threads t1, ..., tn to follow a fixed schedule
3  for(int i = 1; i ⩽ context_bound && loop_cond; ++i){
4     Tid t = ChooseThread(t1, ..., tn);
5     do { // Run next context by thread t
6        RunThreadUntil(t, p_schedule);
7     } while(loop_cond && choose(true, false));
8  }
9  ...... // Interleave each thread sequentially until completion
```

6. Implementation

We have implemented CONCURRIT as an embedded DSL for C/C++. Our implementation is available at http://code.google.com/p/concurrit/. A test in CONCURRIT is simply a C++ program. Our implementation supports the use of CONCURRIT tests in both unit testing and system testing.

6.1 Unit testing with CONCURRIT

For unit testing, the programmer provides the SUT as a shared library with an entry function named test_main. The executable CONCURRIT test loads this shared library and calls test_main multiple times to explore different executions of the SUT.

To instrument the SUT we provide a library with a collection of functions. We give example instrumentation functions below.

```
void concurritThreadStart();
void concurritThreadEnd();
void concurritFuncEnter(void* function, long arg0, long arg1);
void concurritFuncReturn(void* function, long return_value);
void concurritFuncCall(void* caller, void* callee, long arg0);
void concurritMemRead(void* address, int size, long value);
void concurritMemWrite(void* address, int size, long value);
void concurritControlPoint(int pc);
```

CONCURRIT can operate with different event granularities. At one extreme the programmer can insert these functions manually, or one can use our Pin tool [18] for automatically instrumenting (the binary of) the SUT at runtime, capturing all memory operations and function calls/returns. While the the cost can be 10-100x in the former case, the overhead is very small in the latter case. In our experiments with larger programs we needed to insert (manually) very few instrumentation points with negligible cost, which made our approach lightweight.

6.2 System testing with CONCURRIT

We applied CONCURRIT in system testing of three large servers: Memcached, Apache httpd, and MySQL. System tests for these servers carry unique challenges. First, the SUT is composed of many components with nondeterministic behaviors, such as arbitrary network-packet delays and arbitrary assignment of worker threads to connections. Second, the SUT consists of multiple processes interacting with each other using various mechanisms.

To test such servers, first, we run the CONCURRIT test in a separate process and perform the event communication between the SUT processes and the CONCURRIT process using named (FIFO) pipes. Except when specifically blocked and sequentialized by the CONCURRIT test, all the threads in these SUT processes run in parallel. Except for the threads controlled by the CONCURRIT test, all the threads in the distributed processes run in parallel with each other. Second, we define two special events: `TestStart` and `TestEnd` to signal the beginning and the end of an SUT execution. For example, in `memcached`, these events are generated when the requestor process starts and terminates, respectively. The CONCURRIT test only controls the events generated (by the `memcached` server) between `TestStart` and `TestEnd`.

7. Evaluation

In our evaluation of our proposal, we used our implementation of CONCURRIT described in Section 6 and conducted a case study on a collection of multithreaded programs. These programs are from well-known benchmark suits: Inspect [26], PARSEC [3], and RADBench [12]. Each of our benchmarks contains a concurrency bug, either real or manually inserted by existing work [11].

For each benchmark, we followed the testing process that we demonstrated in Section 2 for one of the benchmarks. During this process, we wrote one or more tests in the high-level DSL described in Section 5, aiming at reproducing the bug in the benchmark. We started by writing less exact tests (which are not `SearchAll`) for each bug using partial information about the bug. Based on the information obtained from these test executions we made the tests more precise so that they could reproduce the bug by describing only a small number of key scheduling decisions. We found that we can write concise tests for all of our benchmarks by applying the usage patterns we described in Section 5.1. For small benchmarks we also wrote tests that will search all thread schedules (`SearchAll`) in order to see if model checking could find these bugs efficiently. Moreover, for some benchmarks we managed to write tests that are exact schedules (`ExactSchedule`). The CONCURRIT code for these tests are available at `http://code.google.com/p/concurrit/`. We give sample tests from our benchmarks in Appendix A.

Table 1 shows the results of our study. For each benchmark, the table shows the various tests we wrote to reproduce a bug in the benchmark. Our CONCURRIT tests detected the regarding bugs in all the cases. For each test, we give the number of functions involved in the test, how many extra instrumentation calls were manually inserted in addition to the automated Pin instrumentation, the number of CONCURRIT DSL lines, number of threads involved in the test, and the (average) number of schedules explored before hitting the bug. Note that, in the sixth column we only give the number of threads explicitly controlled by the test; the execution may contain other threads, which are either blocked or executed in an uncontrolled manner. For benchmarks *streamcluster*, *memcached*, *apache-httpd*, and *mysql-server* we did not use Pin and inserted all the instrumentation manually (as reported in the forth column). In this way, our tests become portable as they require no heavy-weight automated instrumentation.

Full model checking. In our study, we were able to apply traditional notion of model checking on small benchmarks, namely bbuf, bzip2, ctrace, pbzip2. The test type "SearchAll" refers to tests of this kind, which was explained in Section 5.1. In these tests, we control the scheduling decision after every memory access (read or write) and function call. We had to instrument all operations that may introduce nondeterminism and change the thread schedule. However, we did not control synchronization operations for locks and condition variables; instead, we inserted scheduling choices before every function call. This gives two benefits. First, it ensures that all model checking tests were able to hit the bug, since instrumenting every function call over-approximates the effect of synchronization-related functions on the thread schedule. Second, it makes our approach *portable*; we do not have to control and implement the semantics of different synchronization operations. Finally, as expected, we should note that full model checking works only for small programs, such a data structures. In our experiments, we bounded the number of executions to $10K$, but even for some small programs `SearchAll` generated more than $10K$ executions.

Customizing the search using information about the bug. While full model checking is impractical, we found that if we use information about a bug to restrict the number of explored schedules in a CONCURRIT test, we can efficiently and reliably reproduce the bugs. As discussed in Section 5.1, we restricted the traditional notion of model checking to customize the search in three aspects: (1) In tests "SearchInFunc", we instrumented only the body of several functions, say $\mathbf{F}$, that we found related to the bug. The third column of Table 1 gives the size of $\mathbf{F}$. Then, we started to search for thread schedules only when threads were executing functions in $\mathbf{F}$. The size of $\mathbf{F}$ highlights a common characteristics of concurrency bugs: they occur due to incorrect synchronization of few functions and can be detected by focusing the search on these functions. (2) In tests "SearchInFuncCB2" and "SearchInFuncCB3", we applied context-bounded model checking techniques [19] with bounds 2 and 3, respectively. (3) In tests "SearchInFuncLS", we scheduled threads at coarse granularity, only after reaching a manually inserted control point. We obtained information about these control points either from the bug report (if the bug report has these details) or from insight obtained from triggering the bug with less exact CONCURRIT tests. For example, to check for atomicity violations using SearchInFuncLS, we inserted these control points around the accesses to the shared variables that we think will involve in the violation, and for ordering violations, we inserted the control points to mark the beginning and end of the code blocks whose different orderings may create an error. We found that, when used in conjunction with restricting the search to functions in $\mathbf{F}$, context bounding and interleaving threads at coarse granularity are quite effective to reproduce the bugs in a small number of executions.

Generating exact buggy schedule. For 8 of our benchmarks, we were able to refine the test to "ExactSchedule", after observing a buggy execution and generalizing its thread schedule in a CONCURRIT test, which guides the execution towards the buggy states. Notice that, compared to tests for search, the size of ExactSchedule tests are not longer. This highlights the fact that in many concurrency bugs only few scheduling decisions involve; by enforcing these decisions, the bug can still happen in spite of the larger, uncontrolled part of the schedule. While sleep-based approaches exploit this fact to effectively reproduce simple concurrency bugs, our technique goes further by making such approches more formal and reliable for reproducing even more complex bugs.

8. Related work

We observe a spectrum of approaches to specifying and controlling the thread schedule in testing of concurrent programs. At one end of the spectrum, there are manual approaches, in which the programmer implements synchronization mechanisms to restrict the possible schedules of threads towards a particular scenario. However, such mechanisms are either ad hoc and unreliable (e.g., in the case of sleep statements) or may require nontrivial and unportable 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

Benchmark	Test pattern	Funcs involved	Manual instr.	Test LOC	Threads controlled	Schedules explored
bbuf	SearchAll	all	0	4	4	20
	SearchInFunc1	2	0	5	4	12
	SearchInFunc2	2	0	4	2	3
bzip2	SearchAll	all	0	4	3	$>10K$
	SearchInFunc	2	0	5	3	510
	SearchInFuncLS	2	3	5	3	8
	ExactSchedule	2	3	6	3	1
ctrace	SearchAll	all	0	4	2	$>10K$
	SearchLargeSteps	4	2	5	2	15
	ExactSchedule	2	2	6	2	1
pbzip2	SearchAll	all	0	4	4	$>10K$
	SearchLargeSteps	3	1	4	4	71
	ExactSchedule	2	1	4	4	1
pfscan	SearchInFunc	1	0	4	2	67
	SearchInFuncLS	1	1	4	2	51
	ExactSchedule	1	1	4	2	1
dedup	SearchInFunc	1	0	4	2	8
	SearchInFuncLS	1	1	4	2	3
	ExactSchedule	1	1	5	2	1

Benchmark	Test pattern	Funcs involved	Manual instr.	Test LOC	Threads controlled	Schedules explored
streamcluster	SearchInFuncLS	all	9	11	4	$>10K$
	ExactSchedule	3	9	11	4	1
mozilla-nspr1	SearchInFunc	1	0	4	2	1020
	SearchInFuncCB2	1	0	6	2	4
mozilla-nspr2	SearchInFuncCB2	2	0	6	2	6
	ExactSchedule	2	1	5	2	1
mozilla-nsp3	SearchSeqFuncs	1	0	5	2	38
spidermonkey1	SearchAll	all	0	4	2	$>10K$
	SearchInFunc	2	0	5	2	$>10K$
	SearchInFuncCB3	2	0	10	2	451
	SearchInFuncCB2	2	0	7	2	226
spidermonkey2	SearchAll	all	0	4	3	$>10K$
	SearchInFunc	4	0	5	3	$>10K$
	SearchInBuggy	4	0	8	3	8
	ExactSchedule	4	0	7	3	1
memcached	SearchInFuncLS	2	12	4	2	10
	ExactSchedule	2	12	4	2	1
apache-httpd	SearchInFuncLS	2	14	4	3	7
mysql-server	SearchInFuncLS	2	9	5	3	30

Table 1. Experimental results. Our CONCURRIT tests detect the bugs in all the cases.

programmer relative to a global timer (ConAn [16], MultithreadedTC [23]) or a sequence of user-defined events expressed in linear temporal logic (IMUnit [9]). While these techniques give the programmer the ability to impose constraints on the interleaving of threads, they do not support 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 [6, 14, 19, 21]. However, these techniques require controlling all sources of nondeterminism in the program, which is impractical for large programs, such as web servers.

Many model checking exploration techniques have been developed that seek to achieve high coverage of executions in a scalable way [13]—such as partial-order reduction [5, 7], symmetry reduction [8], preemption bounding [19], fair stateless model checking [20], and commutativity guided scheduling [25]. These techniques do not directly help to efficiently examine interesting and potentially-problematic interleaving scenarios. Our proposal allows the tester to customize the model checking to target such scenarios. Meanwhile, the tester can still implement these reduction techniques within the CONCURRIT test.

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 [24], probabilistic scheduling [4], and change-aware preemption prioritization [10] 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 [2] 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.

Note that our approach enables rapid prototyping and evaluation of a custom search strategy by writing and exploring a CONCURRIT test. The work in [15] is in the same spirit of ours, but it focuses on Boolean programs and allows the user to describe the model checking algorithm as a compact and high-level fixed-point formulation that can be fed to a general-purpose fixed-point solver.

Recent work in record/replay systems propose recording the thread schedule partially and reproducing the missing parts of the schedule during the replay [1, 22]. These systems do search in order to reproduce a single, feasible schedule, while in our work we do search to enumerate a space of specified schedules.

Acknowledgments

This research supported in part by Microsoft (Award #024263) and Intel (Award #024894) funding and by matching funding by U.C. Discovery (Award #DIG07-10227), by NSF Grants CCF-101781, CCF-0747390, CCF-1018729, and CCF-1018730, and by a DoD NDSEG Graduate Fellowship. The last author is supported in part by a Sloan Foundation Fellowship. Additional support comes from Oracle (formerly Sun Microsystems), from a gift from Intel, and from Par Lab afliates National Instruments, NEC, Nokia, NVIDIA, and Samsung.

References

[1] G. Altekar and I. Stoica. ODR: Output-deterministic replay for multicore debugging. In *SOSP*, 2009.

[2] T. Ball, S. Burckhardt, K. Coons, M. Musuvathi, , and S. Qadeer. Preemption sealing for efficient concurrency testing. *Technical Report MSR-TR-2009-143*, 2009.

[3] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *PACT*, 2008.

[4] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *ASPLOS*, 2010.

[5] P. Godefroid. *Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem.* Springer-Verlag Inc., 1996. URL `citeseer.ist.psu.edu/godefroid95partialorder.html`.

[6] P. Godefroid. Software model checking: The verisoft approach. *In Form. Methods Syst. Des.*, 2005.

[7] G. Gueta, C. Flanagan, E. Yahav, and M. Sagiv. Cartesian partial-order reduction. In *SPIN*, 2007.

[8] R. Iosif. Symmetry reduction criteria for software model checking. In *SPIN*, 2002.

[9] V. Jagannath, M. Gligoric, D. Jin, Q. Luo, G. Rosu, and D. Marinov. Improved multithreaded unit testing. In *ESEC/FSE*, 2011.

[10] V. Jagannath, Q. Luo, and D. Marinov. Change-aware preemption prioritization. In *ISSTA*, 2011.

[11] N. Jalbert and K. Sen. A trace simplification technique for effective debugging of concurrent programs. In *FSE*, 2010.

[12] N. Jalbert, C. Pereira, G. Pokam, and K. Sen. RADBench: A concurrency bug benchmark suite. In *HOTPAR*, 2011.

[13] R. Jhala and R. Majumdar. Software model checking. *In ACM Comput. Surv.*, 2009.

[14] M. Kim, Y. Kim, and H. Kim. A comparative study of software model checkers as unit testing tools: An industrial case study. *In IEEE Trans. Softw. Eng.*, 2011.

[15] S. La Torre, M. Parthasarathy, and G. Parlato. Analyzing recursive programs using a fixed-point calculus. In *PLDI*, 2009.

[16] B. Long, D. Hoffman, and P. Strooper. Tool support for testing concurrent Java components. *In IEEE Trans. Softw. Eng.*, 2003.

[17] S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: Detecting atomicity violations via access interleaving invariants. In *ASPLOS*, 2006.

[18] 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 *PLDI*, 2005.

[19] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *PLDI*, 2007.

[20] M. Musuvathi and S. Qadeer. Fair stateless model checking. In *PLDI*, 2008.

[21] V. Mutilin. Concurrent testing of Java components using Java PathFinder. In *ISoLA*, 2006.

[22] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu. PRES: Probabilistic replay with execution sketching on multiprocessors. In *SOSP*, 2009.

[23] W. Pugh and N. Ayewah. Unit testing concurrent software. In *ASE*, 2007.

[24] K. Sen. Race directed random testing of concurrent programs. In *PLDI*, 2008.

[25] O. Shacham, N. Bronson, A. Aiken, M. Sagiv, M. Vechev, and E. Yahav. Testing atomicity of composed concurrent operations. In *OOPSLA*, 2011.

[26] Y. Yang, X. Chen, and G. Gopalakrishnan. Inspect: A runtime model checker for multithreaded C programs. *Technical Report UUCS-08-004*, 2008.

A. Sample CONCURRIT tests

Parallel tracing library (ctrace)

```
ExactSchedule :
Tid t1, t2 = WaitForDistinctThreads(2, EntersFunc(HASH_READ_ENTER));
RunThreadUntil(t1, AtControl(43) && InFunc(HAS_READ_EXIT));
RunThreadUntil(t2, ReturnsFunc(HASH_READ_ENTER));
RunThreadUntil(t1, ReadsMem &&InFunc(HASH_READ_ENTER));
RunThreadUntil(t2, ReturnsFunc(HASH_READ_EXIT));
RunThreadUntil(t1, ThreadEnds);
```

Online clustering kernel (streamcluster)

```
ExactSchedule :
// Control locations and other events (FuncEnter, FuncReturn) are all manually inserted in functions
// localSearch, localSearchSub, and pkmedian
Tid tmain = WaitForThread(AnyThread, EntersFunc(pkmedian));
RunThreadUntil(tmain, AtControl(42));
Tid t1, t2, t3 = WaitForThread(AnyThread − tmain, AtControl(42));
RunThreadsUntil(t1, t2, EntersFunc(barrier) ‖ ReturnsFunc(pkmedian));
while(InFunc(t1, pkmedian)) {
      RunThreadUntil(t3, EntersFunc(barrier) ‖ ReturnsFunc(pkmedian));
      RunThreadsUntil(t1, t2, t3, ExitsFunc(barrier) ‖ ReturnsFunc(pkmedian));
      RunThreadsUntil(t1, t2, EntersFunc(barrier) ‖ ReturnsFunc(pkmedian));
}
RunThreadsUntil(t1, t2, ThreadEnds);
RunThreadUntil(tm, ThreadEnd);
RunThreadsUntil(t1, t2, ThreadEnds);
```

RADBench Bug 4 (Mozilla NSPR Library – mozilla-nspr1)

```
SearchInFuncCB2 :
Tid t1, t2 = WaitForDistinctThreads(2, EntersFunc(MT_safe_localtime));
Tid t = ChooseThread(t1, t2);
while(choose(true, false)) {
      RunThreadUntil(t, ReadsMem ‖ WritesMem ‖ CallsFunc ‖ ThreadEnds);
}
t = (t == t1) ? t2 : t1;
RunThreadUntil(t, ReturnsFunc(MT_safe_localtime)); // assertion violation!
```

RADBench Bug 5 (Mozilla NSPR Library – mozilla-nspr2)

```
ExactSchedule :
// Control location 42 is in function PR_WaitCondVar
Tid t1 = WaitForThread(AnyThread, EntersFunc(PR_WaitCondVar));
RunThreadUntil(t1, AtControl(42));
Tid t2 = WaitForThread(AnyThread − t1, EntersFunc(PR_Interrupt));
RunThreadUntil(t2, ReturnsFunc(PR_Interrupt));
RunThreadUntil(t1, ReturnsFunc(PR_WaitCondVar));
```

RADBench Bug 6 (Mozilla NSPR Library – mozilla-nspr3)

```
SearchSeqFuncs :
Tid t1 = WaitForThread(AnyThread, EntersFunc(reader_main));
Tid t2 = WaitForThread(AnyThread − t1, EntersFunc(writer_main));
while(InFunc(t1, reader_main) ‖ InFunc(t2, writer_main) {
   Tid t = ChooseThread(t1, t2);
   RunThreadUntil(t, ReturnsFunc(PR_RWLock_Wlock) ‖ ReturnsFunc(PR_RWLock_Rlock)
       ‖ ReturnsFunc(PR_RWLock_Unlock) ‖ ThreadEnds);
}
```

RADBench Bug 2 (Mozilla SpiderMonkey JS – spidermonkey1)

```
SearchInFuncCB3 :
Tid t1 = WaitForThread(AnyThread, EntersFunc(js_GC));
Tid t2 = WaitForThread(AnyThread − t1, EntersFunc(JS_ClearContextThread));
Tid t = ChooseThread(t1, t2);
while(choose(true, false)) {
      RunThreadUntil(t, ReadsMem ‖ WritesMem ‖ CallsFunc ‖ ThreadEnds);
}
t = (t == t1) ? t2 : t1;
while(choose(true, false)) {
      RunThreadUntil(t, ReadsMem ‖ WritesMem ‖ CallsFunc ‖ ThreadEnds);
}
t = (t == t1) ? t2 : t1;
RunThreadUntil(t, ThreadEnds); // segmentation fault!
```

RADBench Bug 8 (Memcached server – memcached)

```
ExactSchedule :
// Control locations (including 42) and other events (FuncEnter, FuncReturn) are all manually
// inserted in functions complete_incr_bin and process_arithmetic_command
Tid t1, t2 = WaitForDistinctThreads(2, EntersFunc(complete_incr_bin)
      ‖ EntersFunc(process_arithmetic_command));
RunThreadUntil(t1, AtControl(42));
RunThreadUntil(t2, ReturnsFunc);
RunThreadUntil(t1, ReturnsFunc);
```

RADBench Bug 15 (MySQL Server – mysql-server)

```
SearchInFuncLS :
// Control locations and other events (FuncEnter, FuncReturn) are all manually inserted in functions
// mysql_insert and MYSQL_LOG::new_file
Tid t1, t2 = WaitForDistinctThreads(2, EntersFunc(MYSQL_LOG :: new_file);
Tid t3 = WaitForThread(AnyThread − t1 − t2, EntersFunc(mysql_insert);
while(InFunc(t1, MYSQL_LOG :: new_file)
    ‖ InFunc(t2, MYSQL_LOG :: new_file) ‖ InFunc(t3, mysql_insert) {
   Tid t = ChooseThread(t1, t2, t3);
   RunThreadUntil(t, AtControl ‖ ThreadEnds);
}
```