Directed Synthesis of Failing Concurrent Executions

Malavika Samak
IISc, Bangalore, India
malavika@csa.iisc.ernet.in

Omer Tripp [*]
Google Inc., Mountain View, USA
trippo@google.com

Murali Krishna Ramanathan
IISc, Bangalore, India
muralikrishna@csa.iisc.ernet.in

Abstract

Detecting concurrency-induced bugs in multithreaded libraries can be challenging due to the intricacies associated with their manifestation. This includes invocation of multiple methods, synthesis of inputs to the methods to reach the failing location, and crafting of thread interleavings that cause the erroneous behavior. Neither fuzzing-based testing techniques nor over-approximate static analyses are well positioned to detect such subtle defects while retaining high accuracy alongside satisfactory coverage.

In this paper, we propose a directed, iterative and scalable testing engine that combines the strengths of static and dynamic analysis to help *synthesize* concurrent executions to expose complex concurrency-induced bugs. Our engine accepts as input the library, its client (either sequential or concurrent) and a specification of correctness. Then, it iteratively refines the client to generate an execution that can break the input specification. Each step of the iterative process includes statically identifying sub-goals towards the goal of failing the specification, generating a plan toward meeting these goals, and merging of the paths traversed dynamically with the plan computed statically via constraint solving to generate a new client. The engine reports full reproduction scenarios, guaranteed to be true, for the bugs it finds.

We have created a prototype of our approach named Minion. We validated Minion by applying it to well-tested concurrent classes from popular Java libraries, including the latest versions of openjdk and google – guava. We were able to detect 31 *real* crashes across 10 classes in a total of 23 minutes, including previously unknown bugs. Comparison with three other tools reveals that combined, they report only 9 of the 31 crashes (and no other crashes beyond Minion). This is because several of these bugs manifest under deeply nested path conditions (observed maximum of 11), deep nesting of method invocations (observed maximum of 6) and multiple refinement iterations to generate the crash-inducing client.

[*] This paper is the result of work that was done while the author was employed by IBM Research.

OOPSLA'16, November 2–4, 2016, Amsterdam, Netherlands
© 2016 ACM. 978-1-4503-4444-9/16/11...$15.00
http://dx.doi.org/10.1145/2983990.2984040

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification—Assertion checkers; D.2.5 [*Software Engineering*]: Testing and Debugging—Symbolic execution, Testing tools

Keywords Dynamic analysis; Client synthesis; Concurrency

1. Introduction

Ensuring the correct behavior of concurrent software is notoriously hard for developers to achieve via testing [33]. Often there are complex scenarios, leading to assertion violations or runtime exceptions, that fall outside the range of behaviors covered by testing. Offending executions typically involve specific input values combined with nontrivial interleaving scenarios [28]. Generating such executions to aid bug detection is challenging.

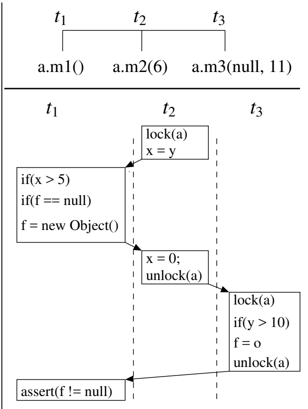
At the same time, there are many available tools for detection of *potential* concurrency bugs, such as data races [13, 45, 46] or atomicity violations [15, 37]. These tools, like developer-written tests, often suffer from limited coverage [44]. Moreover, these tools enforce criteria that may or may not correspond to the developer's notion of correctness. As an example, certain data races may be perceived as benign [35].

In this paper, we focus on the specification of correctness given by the developer either as assert statements or as runtime exceptions (based on the throw statement in the source code). The goal then is to detect, and report, concurrent execution scenarios that lead to such events, which are clear violations of the contract put forward by the program. We study this problem w.r.t concurrent Java libraries, whose code may be executed in unexpected ways by client applications, thereby leading to runtime failures.

Illustrative example. Figure 1(a) presents class X, which defines methods m1, m2 and m3. Method m1 contains an assertion at line 8, which is nested within a condition. Violating this assertion requires a well crafted multi-threaded

```
1   class X {
2    int x = 0;
3    Object f = null;
4    void m1() {
5     if(x > 5) {
6       if (f == null)
7         f = new Object();
8       assert (f != null);
9    }}
10
11   void sync m2(int y) {
12    x = y; ...
13    x = 0;
14   }
15
16   void sync m3(Object o,int y) {
17    if (y > 10)
18     f = o;
19   }
20  }
```

(a)



(b)

Figure 1: Illustrative example. (a): Implementation of class X. (b): The multithreaded test and its execution which violates the assertion.

test that not only ensures that the assertion is reachable but also ensures the assertion is violated. One such test case is depicted in Figure 1(b), where all three methods are invoked by distinct threads with appropriate parameters. The first thread, t_1, invokes method m1 on an instance of class X. Thread t_2 assists thread t_1 to reach the assertion, and thread t_3 assists in failing the assertion. The threads have to follow a specific interleaving for the assertion to fail. One such execution is presented in Figure 1(b), where the execution context switches four times. The intricacies of this example capture the complexities involved in synthesizing a failing execution even for simple examples.

Existing approaches. Static analyses [3, 7, 26, 31] can report potential assertion violations. Unfortunately, these bug reports need to be validated manually for their correctness, which can be a tedious task [27] due to the sheer number of warnings.

Model checking techniques [14, 33] require the presence of bug-exposing multithreaded tests. These tests are executed and the large state space corresponding to distinct schedules is explored to expose the assertion violation. For example, in Figure 1(b), if a multithreaded test with three threads, that makes appropriate invocations is provided, then these techniques can explore the schedules to expose the underlying assertion violation. Dynamic symbolic execution techniques [21, 29] also require the presence of multithreaded tests to detect bugs or their root causes.

A simple yet effective alternative to expose thread-safety violations is to automatically generate random multithreaded tests [38]. However, blind search is infeasible in practice due to the magnitude of the search space and the subtle requirements to trigger the bug, as illustrated above. Synthesizing tests to expose concurrency bugs [43, 44] does not provide a systematic means to work through nested path conditions and interleavings, and also suffers from false positives. For instance, 25% of the 307 reported races in [44] are benign.

This motivates the design of a *directed* approach to synthesize concurrent executions, even in the *absence* of carefully crafted multithreaded tests, such that error situations are systematically searched for by the testing system.

Our approach. As an initial step toward directed detection of concurrency violations in library implementations, we present Minion, a system that — to our knowledge — is the first to simultaneously account for both the input and the interleaving scenario in detecting, and demonstrating, concurrency-induced errors. Both the input and the schedule are essential for full reproduction of a bug, which also guarantees that all warnings reported by Minion are true warnings. There are *no* false positives.

Minion tackles the inherent difficulty of forcing an unintended runtime violation under a concurrent schedule by modeling this challenge as a planning problem. The goal state is to reach a throw statement either due to an assertion violation[1] or due to an undeclared runtime exception. There are multiple possible strategies toward this goal in general.

Minion, starts from the provided client (e.g., a unit test), and iteratively refines it until either a preset budget is exhausted or the target violation is triggered. The provided client can be either a manually written test or generated using automatic test generators [16, 36]. The refinement steps are driven by sub-goals (e.g., to satisfy the condition at line 5, line 12 needs to be executed), computed via static analysis of the library's code, that contribute toward the ultimate goal of failing the assertion.

Minion utilizes three components: (i) static analysis to identify (sub)goals toward the goal of failing the program; (ii) concrete clients to iteratively refine according to new goals; and (iii) constraint solving as a means to combine the path traversed dynamically and the new goal set by the static analysis. At each iteration, a client is executed; the static analysis advises how to refine it; and a new client is synthesized by modeling the current execution trace along with the new refinement goal encoded as constraints, where inputs and interleaving options are treated as free variables. This process resumes until either the target error (or exception) is triggered or testing budget is exhausted.

We defer technical details to future sections, and suffice for now with an intuitive view of the process with reference to the example in Figure 1. Minion accepts as input the implementation of X and a client (e.g., a sequential test that invokes the public class methods with random parameters). Intuitively, the first goal is to reach the assertion. This leads to the synthesis of a client involving both m1 and m2, invoked from two threads. This lets us visit the assertion, provided the value assigned at line 12 is greater than 5 and line

[1] An assert b statement compiles into if (!b) throw ... block.

Tool	Scope	Output	SMT-based	Goal-oriented	Approach
ConTeGe [38]	Libraries	Client programs+input	No	No	Random client generation with linearizability checks
Omen [41]	Libraries	Client programs+input	No	Yes	Targeted test generation for detecting deadlocks
Narada [44]	Libraries	Client programs+input	No	Yes	Targeted test generation for detecting races
Intruder [43]	Libraries	Client programs+input	No	Yes	Targeted test generation for detecting atomicity violations
ESD [53]	Client programs	Input+schedule	Yes	Yes	Execution synthesis via static pruning of execution space followed by symbolic execution
Cortex [30]	Client programs	Input+schedule	Yes	Yes	Exploration of schedule dependent branches by fuzzing the order of events and verifying feasibility
Minion	Libraries	Client programs+input+schedule	Yes	Yes	Iterative test refinement using static analysis and symbolic execution to generate crashing executions

Table 1: Comparison to related work.

5 reads this value. However, due to the object assignment at line 7, the assertion is valid. The next goal, therefore, is to find a means to set f to null. This leads to the invocation of m3 from a separate thread in the client, as guided by the static-analysis component. Further, the parameters provided to m3 must ensure the reachability of the assign statement at line 18, which may write a null value to field f. This value is to be read by the assertion at line 8, and if this value is null, then the assertion will be violated. Needless to say the thread interleaving has to be monitored at every step to ensure that appropriate values are read by relevant accesses.

Intuitively, the above reasoning process — whereby the client is executed dynamically; static analysis guides its refinement; and hybrid constraints, stemming from both the concrete execution trace and the static analysis, are solved to synthesize the next client — is the essence of our approach. This hybrid mode of reasoning, combining static and dynamic analysis in forming the constraints, is a means to tackle classic limitations of both methods in achieving both high coverage and absence of false warnings.

We have experimented with Minion on 10 well-tested classes from 7 popular Java libraries, including the latest versions of openjdk and google guava. The results are highly promising. We have been able to detect 31 concurrency-induced bugs, including previously unknown bugs,[2] within approximately 23 minutes. In contrast, existing approaches [38, 43, 44] are able to cumulatively detect only 9 bugs.

Contributions. The paper makes the following principal contributions:

- We propose a novel design for synthesizing concurrent executions with the goal of triggering unexpected errors, under nested path conditions and thread interleavings, by iteratively refining concrete execution traces.
- We provide detailed description, and explanation, of the algorithms underlying our system and how they interlock. These include runtime monitoring to record the trace, static analysis to identify the intermediate goals towards failing an assertion, as well as use of symbolic constraints extracted from the concrete trace to generate new executions that satisfy pending goals.
- We validate the efficacy of our approach by analyzing real-world concurrent Java classes, including classes declared explicitly to be thread safe. Minion detects 31 concurrency-induced bugs, including confirmed defects in popular libraries that were previously unknown.

[2] Bugs reported in a few JDK classes have been fixed – `http://bugs.java.com/view_bug.do?bug_id=8143394`. Another bug reported by us generated considerable discussion and is marked as a documentation bug.

2. Related Work

There has been significant amount of work in the area of detecting concurrency bugs. Our approach is distinguished from other approaches as we are able to synthesize a failing execution without requiring bug-exposing multithreaded tests or other artifacts like crash dumps as input. Existing directed multithreaded test generation techniques [43, 44] do not consider path conditions and thread interleavings to expose failing executions. We discuss relevant areas of research in turn. In Table 1, we summarize the most closely related tools to Minion, and contrast them with our approach.

Generating multithreaded tests. ConTeGe [38] introduces a testing technique, whereby the generated tests share a common prefix, such that the same class-under-test (CUT) instance is defined for multiple concurrent tasks launched simultaneously. This source of sharing exposes bugs that might be missed otherwise. Additionally, there is an oracle that automatically decides if the observed behavior (crash, deadlock, etc) is a bug based on whether the behavior could still be triggered given any linearization of the involved calls. Our approach does not employ randomization techniques but performs goal-driven iterative test refinement, where we account for multiple shared data structures. As our experimental study indicates, taking these additional steps is of significant practical value.

ConSuite [47] generates unit tests for concurrent classes and attempts to ensure good coverage by exploring different schedules. The magnitude of the search space affects the scalability of these approaches. To address this problem, concurrency-bug-driven multithreaded test synthesis tools [43, 44] are proposed. However, as discussed earlier, handling complex failure scenarios efficiently and accurately is beyond the reach of these approaches.

Constraint-solving-based approaches. Several elegant techniques [20, 25, 28, 30] based on dynamic symbolic execution have recently been proposed that leverage constraint solving. Huang *et al.,* [20] propose an approach for replaying multithreaded executions to reliably reproduce concurrency bugs by recording thread-local execution paths online and encoding execution constraints (e.g. memory order, read-write constraints, etc). Light [25] performs additional optimizations based on intra-thread and inter-thread flow dependencies to perform thread-local recording. Symbiosis [28] reorders events in a failing schedule to produce an error-free schedule, and reports the ordering/flow differences to aid debugging. Broadly, these approaches employ constraint solving to reproduce a bug or to detect its root cause. In contrast, we use constraint solving to construct crash-inducing clients.

In Cortex [30], the order of events from the initial non-failing concurrent execution of a program is perturbed, by using heuristics such as flipping branches near the assertion, to influence schedule-dependent branches. The feasibility of the modified execution is verified using an SMT solver. Though our approach also leverages a concrete trace to mutate in each iteration, there are two key differences. First, our approach targets libraries, and therefore needs to handle data in the form of parameter, variable and field values. Second, our approach is guided by goal refinement rather than locality-based heuristics.

Dynamic analyses and testing. Several different analyses have been proposed that monitor the execution to report data races [8, 13, 24, 45, 46, 48], deadlocks [23, 42] and atomicity violations [4, 15, 37]. However, such bugs might not violate the developer's notion of correctness [35]. Portend [24] is used for detection and classification of data races to analyze the effect of a bad behavior after the fact. Adversarial scheduling approaches [11, 12] employ constraint solving to detect data races and deadlocks in an application. NeedlePoint [34] systematically explores schedules to detect concurrency bugs. Testing frameworks [22, 40] enable the tester to specify a schedule for carefully crafted tests. Minion integrates nicely with these frameworks via the client programs it generates.

Model checking. Model checking is used to verify a given specification, and has been successfully adapted to verify concurrent software [14, 19, 33]. Chess [33] is a model checking tool that explores possible schedules to expose concurrency bugs. Flanagan and Godefroid [14] employ partial order reduction to verify multi-threaded programs. Huang [19] proposes an approach to explore the state space with a minimal number of executions. All of these approaches require the presence of effective multithreaded tests as input.

Solutions based on planning. The objective of ESD [53] is to automatically reproduce crashes in closed programs. This approach explores executions of the target system using a symbolic execution engine (Klee [5]), where static analysis is performed to identify critical control flow edges necessary to reproduce a failure. This reduces the space of possible executions. Other heuristics are enabled at runtime to prioritize the selection of candidate executions. There are key differences from our approach. Conceptually, Minion iteratively refines a particular trace, whereas ESD begins from the set of all possible executions and applies pruning. An advantage of our approach is that we make careful and selective use of static analysis, thereby steering away from its inherent limitations. Beyond that, our approach drives execution towards a crash without prior knowledge of faults in the library, and in particular, without relying on crash-dump information.

In the area of testing, XssAnalyzer [50] is a feedback-driven algorithm for detection of cross-site scripting (XSS) vulnerabilities in websites. It refines the test according to feedback from previous failed attempts to penetrate the website's defenses. In the area of code synthesis, Prountzos *et al.* [39] describe a system, based on automated planning, to synthesize efficient concurrent implementations of graph algorithms that are provably correct.

Automatic sequential test generation. In their seminal work, Godefroid *et al.* [18] propose *concolic* testing, integrating symbolic execution with concrete trace information to enhance path coverage and expose bugs in sequential programs. This is further extended to handle inter-procedural testing [17]. Klee [5] demonstrates the practical impact of automatic test generation on previously well-tested libraries. The Exe tool [6] embodies a comprehensive approach to detect bugs by tracking symbolic constraints on memory accurately. Many automatic test-generation tools [16, 36, 49] for Java utilize feedback from generated tests. Dsdcrasher [9] runs a combination of static and dynamic analyses to automatically crash sequential programs. Our approach is inspired by the effectiveness of these tools, and in particular by the benefits enabled by hybrid reasoning, and is geared towards detecting concurrency bugs.

3. Technical Overview

In this section, we walk the reader through a high-level overview of the Minion system with the help of an example.

3.1 The Minion Architecture

Figure 2 presents the Minion architecture. The Minion workflow assumes as input (i) a concurrent library, (ii) one or more clients of the library (themselves either sequential or concurrent) to be processed serially and (iii) a specification of correctness in the form of an `assert` (or `throw`) statement.

Each iteration of the testing process then consists of three steps:

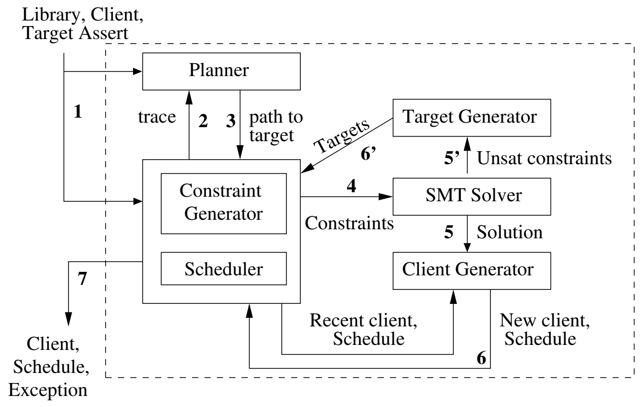


Figure 2: Architecture of Minion. The edge numbers 1...7 represent the workflow. 5',6' is an alternate path to 5,6. The loop 2...6 can execute multiple times.

1. running the current test (initially the provided client);
2. assuming the test fails to trigger the error, computing a target goal for the test to satisfy in the next iteration; and
3. revising the test to meet that goal.

These steps fall under the responsibility of the components described below.

Runtime execution of the client is governed by the *scheduler* component, which runs the given client by following the specified schedule. (Initially the schedule is unconstrained.) The resulting execution trace is recorded for processing by the *constraint generator*, which extracts from it relevant constraints such as the order of execution of certain statements.

Further constraints are obtained through the *planner*. The *planner* examines the trace, and decides how to revise it such that progress is made towards reaching the target goal. The *planner* computes the path conditions leading to the given progress goal and stipulates that these are met.

In sum, the *constraint generator* accepts as its specification (i) the execution trace, as well as (ii) the goal-directed requirements computed by the *planner*, and encodes these as a constraint system. The resulting constraints are then discharged to an off-the-shelf constraint solver.

A solution to the constraints is, in essence, an input/schedule configuration that is guaranteed to generate an execution trace meeting the goal put forward by the *planner*. The *client generator* uses this solution, plus the most recently generated client, to create a fresh client. Failure to find a solution for the constraint system results in the generation of alternative targets (or sub-goals) by the *target generator*. If the *target generator* fails to find such a target, then the solving loop is reset to enable pursuit of another plan. The solving process proceeds until either the target is reached or a predetermined testing budget is exhausted. Upon successfully reaching the target, the refined client and its associated schedule are output by our system.

This style of reasoning is not complete, yet it is fully sound. That is, any warning reported by Minion is guaranteed to be true.

3.2 Illustrative Run

Figure 3 illustrates the dynamics of Minion on a running example, which we shall use throughout the rest of the paper. It presents the class under test X, the input test p0, and the tests p1, p2, p3 generated by Minion. The generated part of the tests is demarcated in the figure. For ease of presentation, we skip the code corresponding to the creation of multiple threads as well as for object sharing across threads. Methods that need to be executed by newly created threads are also specified. The remaining instructions including the code given in the input test are executed by a default *main* thread.

The schedule presented below each test in the figure specifies the scheduling constraints that need to be followed. For example, (t2, 12), (t1, 5) means that line 5 can only be executed by thread t1 after line 12 is executed by thread t2. We now discuss the application of Minion to X.

Here, the class under test X consists of methods m1 through m3, and the client is the (sequential) p0 method. The first step in applying Minion is to identify the f != null assertion at line 8 as the primary target. This is either done by the planner or can be an input given by the developer.

In the first iteration, the (sequential) p0 method is executed. The execution skips the body of the condition at line 5, which the planner identifies as necessary to reach the assertion. Hence, a goal is defined to have $x > 5$ at line 5. The constraint generator uses this goal and the observed trace to generate the appropriate constraints. Feeding these constraints into the solver, where inputs and interleavings are modeled as free variables, yields a concurrent variant of p0, such that

- from the schedule perspective, the executions of m1 and m2 are interleaved with the body of m1 occurring in between x = y and x = 0; and
- from the input perspective, the argument to m2 is 6 rather than 0, and m1 and m2 are now invoked on the same receiver object (s1).

The resulting client, p1, is shown in Figure 3(c). The execution of the client with the schedule is shown in Figure 4(a). It reaches the assertion, but there is no violation of the assertion. Thus, a new goal is defined, which is to reach line 8 such that $f \equiv null$. Minion thus searches for an alternative goal, which is identified as the field assignment statement at line 18. The corresponding constraint is $y > 10$. Hence, the current version of the client is revised further to concretely expose a different assignment to the field f, such that from

```
1  class X {
2   int x = 0;
3   Object f = null;
4   void m1() {
5    if(x > 5) {
6      if (f == null)
7        f = new Object();
8      assert (f != null);
9   }}
10
11  void sync m2(int y) {
12   x = y; ...
13   x = 0;
14  }
15
16  void sync m3(Object o,int y) {
17   if (y > 10)
18    f = o;
19  }
20 }
```

(a)

```
Iteration 1:

p0() {

/* input test */
  X a = new X();
  X b = new X();
  a.m1();
  b.m2(0);
  a.m3(b,0);
}

Schedule:
Nil
```

(b)

```
Iteration 2:

p1() {

/* input test */
  X a = new X();
  X b = new X();
  a.m1();
  b.m2(0);
  a.m3(b,0);

/* generated part */
  X s1 = new X();

/* set up threads
   t1,t2 */
  ...

/* execute in t1 */
  s1.m1();
/* execute in t2 */
  s1.m2(6);
}

Schedule:
(t2,12),(t1,5),(t2,13)
```

(c)

```
Iteration 3:

p2() {

/* input test */
  X a = new X();
  X b = new X();
  a.m1();
  b.m2(0);
  a.m3(b,0);

/* generated part */
  X s1 = new X();
  X s2 = new X();
  X s3 = new X();

/* set up threads
   t1,t2,t3 */
  ...

/* execute in t1 */
  s1.m1();
/* execute in t2 */
  s1.m2(6);
/* execute in t3 */
  s2.m3(s3,11);
}

Schedule:
(t2,12),(t1,5),(t2,13)
```

(d)

```
Iteration 4:

p3() {

/* input test */
  X a = new X();
  X b = new X();
  a.m1();
  b.m2(0);
  a.m3(b,0);

/* generated part */
  X s1 = new X();

/* set up threads
   t1,t2,t3 */
  ...

/* execute in t1 */
  s1.m1();
/* execute in t2 */
  s1.m2(6);
/* execute in t3 */
  s1.m3(null,11);
}

Schedule:
(t2,11),(t2,12),(t1,5),
(t2,13),(t2,14),(t1,7),
(t3,16),(t3,18),(t1,8)
```

(e)

Figure 3: Illustrative example. (a): Implementation of class X. (b): p0 is an initially provided client that invokes methods in X. (c): p1 is an intermediate client, constrained by schedule shown below it, where t1 satisfies conditional at line 5. (d): p2 is an intermediate client where t3 exposes the write at line 18. (e): p3 is the final multithreaded client that exposes an assertion violation at line 8 when the schedule shown below it is followed. In (c), (d) and (e), the code for creation of threads and passing the relevant objects to them is elided for ease of presentation.

the input perspective, m3 is invoked with 11 as its second argument.

The resulting client, p2, is shown in Figure 3(d). The corresponding execution is given in Figure 4(b). It exposes the write at line 18, but does not necessarily push toward the assertion violation at line 8 because the field write is on a different object (s2) and the assignment is non null. This is because the solver is not constrained yet to use the same receiver object for m3 as is used for m1 and m2. Thus, the client is revised further, such that

- from the schedule perspective, the execution of m3 is interleaved in between lines 7 and 8; and
- from the input perspective, m3 is invoked with null as its first parameter on the same receiver object (s1) as in t1 and t2.

The revised client p3, shown in Figure 3(e) when executed on the generated schedule appearing below it, induces an assertion violation. This corresponds to the execution shown in Figure 1(b).

In summary, given an implementation of a class X and an initial client p0, Minion is able to generate a client p3 along with a schedule that exposes an assertion violation in X.

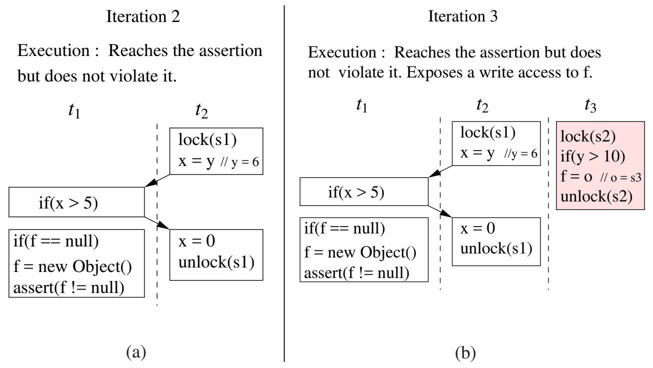


Figure 4: Execution of clients for illustrative example. Shaded region denotes operations on a different object.

4. Design

We now discuss in detail the design of our approach, and elaborate on the various components. The main analysis algorithm accepts as input a program p_0 as well as an initial target tgt_0. The target is a `throw` statement to be visited, which — as noted above — also covers assertions.

The program p_0 is the provided client that invokes library methods. For ease of explanation, we consider each library method invocation by p_0 to be from a distinct thread. In this setting, given a sequential program p_0, we treat p_0 as the serial composition of different threads that each execute a different library call. For instance, the sequential client shown in Figure 3(b) can be seen as invoking m1, m2 and m3 via three distinct threads; the execution is ordered by the scheduler, such that first m1 executes, then m2, and finally m3.

Algorithm 1 The MINION algorithm

```
1:  procedure SOLVE($p_0$, $tgt_0$)
2:      TS ← [$tgt_0$]                                   ▹ initialize target stack
3:      p ← $p_0$; s ← initial order
4:      while has budget do
5:          $\tau$ ← EXECUTE(p, s)                     ▹ Execute program/schedule pair
6:          ($[t \mapsto \overline{\rho}]$, fin) ← PLAN(p, $\tau$, top(TS))   ▹ set per-thr. pc.s
7:          $\phi$ ← ENCODE(p, $\tau$, $[t \mapsto \overline{\rho}]$)     ▹ encode as constraints
8:          (p′, s′) ← CSOLVE($\phi$, p)               ▹ invoke solver
9:          if (p′, s′) = ⊥ then                       ▹ solver failure
10:             tgt ← NEXTTGT(p, $\tau$, top(TS))
11:             if tgt = ⊥ then              ▹ no new target ⇒ dead end
12:                 TS ← [$tgt_0$]                     ▹ reset target stack
13:                 (p, s) ← ($p_0$, initial order)    ▹ reset (p, s)
14:             else
15:                 TS ← TS :: tgt             ▹ push subgoal onto stack
16:             end if
17:         else
18:             (p, s) ← (p′, s′)        ▹ update program/schedule pair
19:             if fin then
20:                 TS ← pop(TS)                       ▹ target was satisfied
21:             end if
22:             if TS = [] then ▹ TS is empty ⇒ $tgt_0$ has been met
23:                 return (p, s)                      ▹ success!
24:             end if
25:         end if
26:     end while
27:     return ⊥
28: end procedure
```

4.1 The MINION Main Loop

Algorithm 1 presents the main loop that iterates until it either succeeds in meeting tgt_0 (line 23), or it runs out of budget (loop test). We generally denote failure by a ⊥ return value (line 27). Within a given iteration, the current program/schedule pair is executed using the procedure EXECUTE to obtain a trace τ. That trace, together with the program p and the current target represented by the top of the target stack (TS), are fed into the planning procedure PLAN.

The PLAN procedure outputs a *specification* containing the path conditions that each thread should traverse. fin denotes whether the *current* target is achieved. This is not necessarily the case, since PLAN may form the specification w.r.t. the current trace τ, which may not reach the current target (e.g., the trace obtained by executing p0 in Figure 3(b)). We return to PLAN below, in Section 4.3. For now, we note our assumption that PLAN is non-deterministic. That is, given a program p, trace τ and target tgt, PLAN chooses nondeterministically between the available plans. This is important as a mechanism to enable fail/retry progress as is explained subsequently.

The next two steps are to encode the specification as set ϕ of constraints (via ENCODE) and then discharge ϕ to a constraint solver (the CSOLVE procedure). If the solver fails to synthesize a program/schedule pair meeting the specification (i.e., if tgt = ⊥), then a sub-goal is selected that is speculated to facilitate satisfaction of the current target (via NEXTTGT).

If this attempt fails, then we perform full backtracking. The solving loop is restarted with (i) a fresh target stack including only tgt_0 (line 12) and (ii) the original program p_0 together with the initial ordering (line 13) that is dependent on the provided client. Here, the non-deterministic contract of PLAN enables (if possible), with non-zero probability, a different unfolding of the refinement process.

Alternatively, if the solver succeeds, then the new program/schedule pair (p′, s′) is fixed as the current pair (line 18). Furthermore, if this pair satisfies a plan that meets the corresponding target (i.e., fin is true), then that target has effectively been satisfied, and consequently it is removed from the target stack (line 20).

If the target stack becomes empty, then the main analysis loop has terminated successfully. Otherwise, we loop back to line 5 to execute the revised program/schedule pair, such that the resulting trace τ should bring us closer to the current target, and so ultimately towards a true value for fin.

In the following subsections, we dive into the procedures invoked by the main loop. These are EXECUTE, PLAN, ENCODE, CSOLVE and NEXTTGT. We discuss each of them in detail.

4.2 The EXECUTE Procedure

Algorithm 2 clarifies how we execute the program p under a specified schedule s. The schedule is simply a queue specifying the dependencies among the instructions of different threads.

A random thread from the *active* set of threads is selected to execute (line 4), and the next instruction from that thread is identified (line 5). If the thread/instruction pair is present in the schedule s, denoting dependencies pertaining to its execution, then we ensure that the pair is in the front of the queue (line 7) so that all the instructions it depends on have already been executed. Accordingly, the instruction is executed and the pair is removed from s (lines 8 and 9).

On the other hand, if it is not in the front of the queue, then some dependencies still remain to be executed. Therefore, we move the thread to *blocked* (line 12). Moreover, other instructions from the same thread in s that occur before the instruction under consideration, due to program order (represented using $\prec$ in the algorithm), are also removed from s (line 11).

Algorithm 2 The EXECUTE procedure

```
1: procedure EXECUTE(p, s)
2:     $blocked \leftarrow \emptyset$; $active \leftarrow$ all threads
3:     while $active \cup blocked$ is not empty do
4:         $t \leftarrow$ pick randomly from $active$
5:         $instr \leftarrow$ next instruction from $t$
6:         if $(instr, t) \in$ s then
7:             if $(instr, t) =$ front(s) then
8:                 Execute $instr$
9:                 s $\leftarrow$ s $- \{(instr, t)\}$
10:            else
11:                s $\leftarrow$ s $- \{(x, t) \mid (x, t) \prec (instr, t)\}$
12:                Move $t$ from $active$ to $blocked$
13:            end if
14:        else
15:            Execute $instr$                ▷ unconstrained by s
16:        end if
17:        $(x, t') \leftarrow$ front(s)
18:        Ensure $t'$ is in $active$
19:    end while
20: end procedure
```

Note that not all path conditions can be modeled by the constraint solver, and so the client might diverge from its intended path according to PLAN. If this happens, then instructions that were scheduled to execute but were skipped due to the path change are simply removed, while instructions from other branches are simply executed unconstrained (line 15).

Subsequently, we make sure that the thread at the front of s is in *active* so that it can be scheduled. The generated trace (τ) is used by subsequent procedures.

Illustration. In the running example, the EXECUTE procedure is invoked in each of the four iterations. In the first iteration, it receives the sequential client p0 and an empty schedule, depicted in Figure 3(b) . The execution of this client will be unconstrained by EXECUTE, as the schedule is empty. In the second iteration, EXECUTE receives a non-empty schedule and a multi-threaded client p1, depicted in Figure 3(c). Both threads t_1 and t_2 are allowed to execute freely until the relevant instructions in m1 and m2 are executed. When thread t_1 attempts to execute the instruction at line 5, the EXECUTE procedure checks if thread t_2 has executed the instruction at line 12. If thread t_2 is yet to execute this instruction, thread t_1 will be blocked until thread t_2 executes it. Similarly when thread t_2 attempts to execute the instruction at line 13 it will get blocked if thread t_1 has not yet executed the instruction 5. Threads are allowed to execute an instruction without any checks if the instruction is not part of the schedule. For instance, thread t_1 is allowed to execute the instructions at line 6,7 and 8 freely. The executions of the clients in the third and fourth iterations happen in a similar manner.

4.3 The PLAN Procedure

As explained earlier, the functionality carried out by PLAN is to decide which path (or rather, path prefix) should be followed by each of the threads. PLAN returns a pair. The first component is the mapping from threads to branching decisions, except that the first incorrect branching decision made by the thread being driven to a target is now negated. The second component is fin. We provide a pseudo-code description of this procedure in Algorithm 3.

The first step is to partition the input trace τ by thread identifiers (line 2), such that each thread t incident in τ is mapped to the sequence of branching decisions it executed. Subsequently, we identify (at line 3) the thread t' that is assigned to reach the target as well as the method invocation containing the target.

Algorithm 3 The PLAN procedure

```
1: procedure PLAN(p, $\tau$, tgt)
2:     $T \leftarrow [t \mapsto$ pc $(\tau, t)]$            ▷ partition pc.s by threads
3:     $t' \leftarrow$ thread[tgt]; m $\leftarrow$ method (tgt)
4:     $\rho_{1...n} \leftarrow$ reach (m, tgt)          ▷ statically derive pc.s
5:     if $\rho_{1...n} \not\sqsubseteq T[t']$ then        ▷ not a subsequence in trace
6:         Find min $i$ in $[1 \ldots n]$ $s.t$ $\rho_{1...i-1}.\neg\rho_i \sqsubseteq T[t']$
7:         $\overline{\rho}_{t'} \leftarrow$ prefix $(T[t'], i)$      ▷ get prefix from trace
8:         return $(T[t' \mapsto \overline{\rho}_{t'}.\rho_i]$, false)     ▷ flip cond at $i$
9:     end if
10:    return $(T$, true)
11: end procedure
```

If there is no such previous assignment (e.g., in the first iteration), then we use static analysis to identify the method containing the target, and obtain the thread(s) invoking the method. Among this set, we choose a thread that is unassigned for other purposes. This thread is assigned the responsibility to reach its respective target, in future iterations. In the running example, thread t1 is responsible for reaching the assertion violation at line 18 from the second iteration onwards.

We use static analysis to specify the path conditions $\rho_1 \ldots \rho_n$ that are necessary to reach the target tgt in m at line 4. For brevity, we use the notation $\rho_{1...n}$ to mean $\rho_1 \ldots \rho_n$. The static analysis employed here is a naive (control) reachability analysis that elides precise static reasoning about path conditions.

If the set of conditionals is a sub-sequence of the trace corresponding to t' (denoted $\rho_{1...n} \sqsubseteq T[t']$), then the target is already reachable. We use sub-sequence for comparison because there can be conditionals in the concrete trace that can take any value and do not affect reachability of the target. In this case, we return the specification T with the fin flag set to true (at line 10).

Otherwise, if the path conditions are not a sub-sequence, then we find the minimum i, such that the branching decisions made by t' up to ρ_{i-1} form the prefix of a path leading to the current target. The subsequent sequence of conditionals ($\rho_i \ldots \rho_n$) is not currently satisfied, and ρ_i is the earliest point starting from which execution diverges away from the target `tgt`. We obtain the prefix of conditionals from the trace for t' upto, but not including, ρ_i (at line 7).

Here, the prefix ($\overline{\rho}_{t'}$) includes conditionals along the path to the target that do not affect its reachability (i.e., the target is reachable independent of the evaluation of the conditional). We enforce this so that future iterations of the client traverse previously executed paths to reach the target. Otherwise, following a different path towards the target can modify the data values, which can adversely affect key conditionals along the path leading to the target.

The specification for t' is $\overline{\rho}_{t'}$ concatenated with the flipped conditional, ρ_i. The `fin` flag is set to `false`.

Illustration. The Plan procedure provides the necessary specifications that can be used by the Encode procedure. We illustrate the working of Plan in the first iteration. After the sequential client `p0` completes its execution, the Plan procedure is invoked. The initial target, which is the assertion violation at line 8, and the trace corresponding to the execution of `p0` are the inputs to Plan in the first iteration.

As a first step, Plan checks whether a thread has been assigned to reach the target. Since this is the first attempt to reach the target, there are no assignments. Therefore, Plan assigns a new thread t_1 to reach the target. Next, it identifies that the execution of `p0` diverged away from the target at line 5 in class X. Therefore, it specifies that the corresponding condition needs to be flipped by the new thread t_1 (lines 5–8 in the algorithm). This specification is returned to the main algorithm. The job of generating the relevant constraints, solving them and generating a refined client is performed by the other components.

4.4 The Encode Procedure

The Encode procedure accepts as input the current client `p`, the trace τ obtained by executing the program according to the given schedule and the specification returned by the Plan procedure. It is assigned the task of returning the constraints, which are used by subsequent procedures to refine the client suitably.

Encoding of the constraints is broadly based on earlier approaches [20, 28, 52] that are used to debug/reproduce a failing concurrent execution. However, unlike these approaches, our goal is to enable synthesis of a failing concurrent execution. In comparison to MCR [19], which employs constraint encoding to synthesize a failing execution for closed programs based on specific thread schedules, our approach needs to handle other challenges: (a) we need to encode path conditions to enable branching, and (b) we need to fix the parameters to public methods invoked by the library client (without the context that closed programs provide to constrain such parameters). These are key differences in the requirements from our analysis.

Most importantly, operations (read, write, lock) can be performed on distinct memory locations, and our analysis should recognize the possibility of these locations aliasing. Also, the final (crashing) execution may involve more threads than the initial client (i.e., the system should support synthesis of concurrent executions with more threads than observed). Moreover, the current execution might not cover the required targets. Based on the input from the planner, the constraint system should generate constraints that can be used to generate new clients that are able to make progress towards the target. These challenges need to be handled by the Encode procedure.

Symbolic Trace. Initially, a symbolic trace of the events while executing a library method is recorded. We use the concurrent SSA form [52], a variant of the classic SSA form [32], to distinguish the variables in the execution. We use this form to account for the fact that reads need not flow from the latest write in the control flow, but can flow from writes unrelated to the flow due to concurrency. For instance, in Figure 3, `x` at line 5 in `m1` can be defined by the assignments in `m2`. Consequently, *fresh* symbolic values are assigned to these reads. Unlike classic SSA, the concurrent SSA form necessitates the fresh assignments to handle concurrency. Note that the recorded trace provides concrete points-to information, based on which we determine aliasing between memory locations precisely.

Instruction	Action
X a = new X()	-
X b = new X()	-
m1()	$e_1 : this_0@0 \mapsto p_0, this_0.x_0@0 \mapsto p_1,$ $this_0.f_0@0 \mapsto p_2$
if(x > 5)	$e_2 : this_0.x_1@0 \leq 5$
m2(int y)	$e_3 : this_0@1 \mapsto p_3, this_0.x_0@1 \mapsto p_4,$ $this_0.f_0@1 \mapsto p_5, y_0@1 \mapsto p_6$
lock(this)	$e_4 : lock(this_0@1)$
x = y	$e_5 : this_0.x_1@1 = y_0@1$
x = 0	$e_6 : this_0.x_2@1 = 0$
unlock(this)	$e_7 : unlock(this_0@1)$
m3(Object z, int y)	$e_8 : this_0@2 \mapsto p_7, this_0.x_0@2 \mapsto p_8,$ $this_0.f_0@2 \mapsto p_9, z_0@2 \mapsto p_{10},$ $y_0@2 \mapsto p_{11}$
lock(this)	$e_9 : lock(this_0@2)$
if(y > 10)	$e_{10} : y_0@2 \leq 10$
unlock(this)	$e_{11} : unlock(this_0@2)$

Figure 5: Symbolic trace.

Figure 5 presents the symbolic trace generated for the execution of the initially provided (sequential) client. We use the notation $var_{version}@id$ to denote the symbols in the trace. It specifies the *version* of *var* in the method invocation represented by *id*. For example, $this_0.x_1@0$ corresponds to reading version 1 of variable `x` at line 5, where the invocation identifier (of `m1`) is 0. Every update to the variable increments its version number.

The variables given by e_i and p_i are *free* variables and correspond to the event identifiers and the parameters supplied to the invocation. Event identifiers are free variables that can enable appropriate scheduling. Parameters are free variables to capture the open-world setting and enable passing of relevant objects as parameters to the invocations in the newly generated client.

Also, a careful reader will notice that even though the receivers for `m1` and `m3` are the same, they are given different identifiers ($this_0@0$ and $this_0@1$). This is because, as discussed in the beginning of Section 4, we consider each library method invocation in the provided client to originate from a distinct thread.

Goal Constraints. The overall set of generated constraints is given by ϕ and is defined as:

$$\phi = \phi_{path} \wedge \phi_{sync} \wedge \phi_{po} \wedge \phi_{rw} \wedge \phi_{param}$$

Here, ϕ_{path} encodes the partial control flow of different threads as specified by the PLAN procedure; ϕ_{sync} corresponds to the constraints related to the synchronization operations; ϕ_{po} specifies the program ordering; ϕ_{rw} encodes the relationships between the reads and writes based on *potential* shareability; and ϕ_{param} provides the constraints pertaining to the invocation parameters. We now describe each subgroup of constraints in detail.

Path Constraints. The PLAN procedure provides a specification that maps each thread to a set of path conditionals. We encode this as part of the path constraints ϕ_{path}. In the current trace τ, we observe that $this_0.x_1@0 > 5$ does not hold (see e_2 in Figure 5). Since, the PLAN procedure specifies that this condition needs to be flipped, we flip this constraint accordingly. Path constraints pertaining to other threads are unmodified (e.g., $y_0@2 \leq 10$). The generated path constraint for the running example is encoded as follows:

$$\phi_{path} = (this_0.x_1@0 > 5) \wedge (y_0@2 \leq 10)$$

Synchronization Constraints. Lock operations in the trace can be on different objects. However, we have to consider the possibility of the objects aliasing. Therefore, for any pair of lock operations, when the lock objects alias, we impose an additional constraint in the form of mutual exclusion of respective critical sections. That is, we impose an ordering between unlock and lock operations. For the running example, lock objects in `m2` and `m3` are specified symbolically as $this_0@1$ and $this_0@2$ respectively. These are either not aliased or the corresponding unlock and lock operations need to be ordered. The generated synchronization constraint for the running example is encoded as follows:

$$\phi_{sync} = (this_0@1 \neq this_0@2) \vee (e_{11} < e_4 \vee e_7 < e_9)$$

Program Order Constraints. These constraints capture the ordering of the various instructions in the execution trace. Since each library method invocation is from a distinct thread, we need to order the events within an invocation only. The generated constraint ϕ_{po} for the running example is given as follows:

$$\phi_{po} = (e_1 < e_2) \wedge (e_3 < e_4 < e_5 < e_6 < e_7) \wedge (e_8 < e_9 < e_{10} < e_{11})$$

Read-Write Constraints. Read and write operations in the original/generated clients can apply to *distinct* concrete locations. However, our approach has to also consider that these operations may be performed on the same object. Hence, we pair the object dereferences in the symbolic trace with appropriate writes. The pairing is derived based on the accessed field. Further, each such pairing should also ensure appropriate ordering of the events such that there is no other intermediate write to the same location. These details are encoded in ϕ_{rw}. A partial listing of the generated constraints for our running example is as follows:

$$\phi_{rw} = ((this_0.x_1@0 = this_0.x_1@1) \wedge (e_5 < e_2) \wedge (e_6 < e_5 \vee e_2 < e_6) \wedge \ldots) \ldots$$

In other words, the read at line 5 can read from the initial assignment at line 2 or the writes in method `m2` at lines 12 and 13, respectively. For the read to be from line 12, other writes (e.g., at line 13) need to happen before line 12 or after the read. This reasoning is applied to all possible read/write pairings. Other constraints not shown above can be derived similarly.

Parameter Constraints. Different fields of the input object are assigned free variables (p_i). Hence, the constraint solver can assign values to them *independently*, which can lead to malformed objects that are inconsistent with properly constructed objects. Therefore, we restrict the assignments of object references as parameters to previously observed objects *only*. We perform a heap walk of all object instances in the client and identify their fields. If a field is of a reference type, then we assign it a unique identifier. Moreover, we encode these constraints to avoid creation of objects with field values from distinct objects. The partial constraints corresponding to ϕ_{param} for the running example are given below:

$$\phi_{param} = (p_0 = a \wedge p_1 = a.x \wedge p_2 = a.f) \vee (p_0 = b \wedge p_1 = b.x \wedge p_2 = b.f) \ldots$$

This denotes that the parameters $p_0 \ldots p_2$ can be due to previously observed objects `a` or `b` in Figure 3. We can also assign `null` values to the parameters. We defer discussion on how this is handled to Section 5.

4.5 The CSOLVE Procedure

The CSOLVE procedure generates a revised client and schedule, which will drive the execution towards the current target, when possible. It takes as input the set ϕ of encoded constraints plus the current client `p`. Algorithm 4 presents the corresponding pseudo-code.

The first step is to discharge the constraints ϕ to an off-the-shelf SMT solver. We use Z3 [10] in our implementation. If the constraints are unsatisfiable, then $\perp$ is returned (lines 3–5).

If a solution σ exists, then σ provides information how to order the instructions of different threads due to the encoded constraints. The sorted order of the instruction/thread pairs (line 6) forms the updated schedule `s`.

Algorithm 4 The CSolve procedure

```
1: procedure CSolve($\phi$, p)
2:     $\sigma \leftarrow$ SMTSolver($\phi$)
3:     if $\sigma$ = unsat then
4:         return $\perp$                    ▷ unsatisfiable constraints
5:     end if
6:     s ← Extract the event ordering from $\sigma$
7:     params ← Extract parameter assignments from $\sigma$
8:     p′ ← gen_client(params, p)
9:     return (p′, s)
10: end procedure
```

The parameter assignments to various objects are available in the solution. These assignments and the current client p are input into gen_client. gen_client refines p to generate a new client, p′. Broadly, our approach to generate clients according to provided constraints is based on existing approaches [41, 43, 44]. Briefly, it corresponds to creation of threads, invocation of various methods from these threads with the specified parameter values. Existing objects from the execution of the initial client can be reused to provide the objects for the various method invocations. For our purposes, it suffices that the generated client along with the schedule satisfy ϕ.

Illustration. We now describe the process of generating client p1 after the first iteration. CSolve receives the constraints encoded by Encode and solves these constraints using a SMT solver. For the current iteration, the solution indicates:

- methods m1 and m2 need to be invoked on the same object,
- the second parameter to m2 is 6.

These requirements are used while generating p1. Further, the schedule to be followed while executing the client p1 is extracted from the solution (line 6). The newly generated client and schedule are returned to the main algorithm.

4.6 The NextTgt Procedure

Procedure NextTgt is responsible for locating a next target to push onto the target stack. It is invoked if the constraint solver is unable to generate a revised client. If the target generator fails, then full rollback is enforced. Otherwise a new target is returned. A pseudo-code description of NextTgt is provided in Algorithm 5.

The first three steps (lines 2–4) are similar to the steps in the Plan procedure. The set of path conditions is partitioned per thread; the thread and method corresponding to tgt are identified. Unlike the Plan procedure, tgt here always has an assigned thread. Static analysis is used to derive the critical path conditions $\rho_{1 \ldots n}$ toward tgt. Next, we find the minimum i, such that the path diverges away from the target in the trace associated with the assigned thread.

We calculate the dynamic slice starting from the relevant conditional ρ_i, and derive the *vars* (line 6). Static analysis is used again to detect the writer statements defining these variables. These are the locations where the variables are updated. The analysis builds a propagation call graph (interleaving pointer analysis and call-graph construction) for the library, and scans it for heap updates. This forms the possible set of (next) targets (*tgts* at line 7).

Algorithm 5 The NextTgt procedure

```
1: procedure NextTgt(p, $\tau$, tgt)
2:     $T \leftarrow [t \mapsto$ pc $(\tau, t)]$
3:     $t' \leftarrow$ thread[tgt]; m ← method (tgt)
4:     $\rho_{1 \ldots n} \leftarrow$ reach (m, tgt)
5:     Find min $i$ in $[1 \ldots n]$ s.t $\rho_{1 \ldots i-1}.\neg\rho_i \sqsubseteq T[t']$
6:     $vars \leftarrow$ dslice($\tau, \rho_i$)
7:     $tgts \leftarrow$ writers($vars$)
8:     if $\exists$ tgt′ $\in tgts$ s.t. tgt′ $\notin$ used($\rho_i$) then
9:         used($\rho_i$) ← used($\rho_i$) $\cup$ {tgt′}
10:        return tgt′
11:    end if
12:    Clear all used sets.
13:    return $\perp$
14: end procedure
```

For the conditional under consideration (ρ_i), we choose a target tgt′ that is not already used by the current exploration. This selection is non-deterministic to enable exploration of other targets in future iterations, if the current plan fails. We return this target after inserting it into the used set.

If no target is available, then the result of NextTgt is $\perp$ after clearing all the used sets (line 12), which corresponds to the rollback performed by the main loop. This, complemented by random selection of the target at line 8, enables exploration of a different sequence of targets across different iterations of the main solver loop in Algorithm 1.

Illustration. For the running example, the NextTgt procedure is invoked once. After the second iteration, the Plan procedure specifies that the condition at line 8 is flipped by modifying the client p1. Other components of our approach fail to generate such a client by using the execution of p1. Since it is not possible to make progress to reach the current target, NextTgt procedure is invoked to provide alternate targets so that the original target can be reached.

NextTgt analyzes the execution of client p1 and identifies that the execution failed to flip the condition at line 8. A data-flow slice of this execution is obtained to identify the variables that influence the outcome of this conditional (line 6 in the algorithm). In this case, the conditional is only dependent on the value of field f. Therefore, the procedure searches for statements that can modify the value of field f (line 7 in the algorithm). There are two statements in class X that modify the field f (lines 7 and 18). As the execution has already covered the write of f at line 7 and because it was not useful in flipping the condition, we discard it. The other target at line 18, which has not been explored, is returned as the next target.

The other components (Plan, etc) now attempt to reach this new target so that the original goal of violating the assertion can be achieved.

4.7 Limitations

The Minion design is driven by practical considerations, informed by our experiences in developing/debugging/testing concurrent Java libraries. We conclude this section with a discussion of the limitations of our design.

First, selection of targets follows a greedy strategy. Therefore, earlier choices can lead to generation of clients that evolve away from, rather than towards, the target violation (even if the violation is feasible). We address this via full rollback (lines 12-13 in Algorithm 1). Non-determinism in target selection increases coverage, though there is no guarantee regarding the complexity/efficiency of the process.

Second, if the path conditions contain constraints that defeat the expressive power of the supporting solver and alternative sub-goals are absent to satisfy a conditional, then our design will fail to expose the violation. This is a general limitation of solver-based testing/analysis approaches.

Finally, our approach is limited in handling scenarios wherein sub-goals are not merely satisfied by reachability, but require a more complicated condition. For example, if an instruction needs to be executed a specific number of times for a condition to be satisfied, then it would fall outside the reasoning power of our current prototype system.

Notwithstanding these issues, our experimental results on real Java libraries demonstrate the effectiveness of the proposed design in discovering bugs that manifest under complex input conditions and interleaving scenarios. (See Section 6.)

5. Implementation

In this section, we highlight design choices of interest w.r.t. the implementation of our system.

Frameworks. We have implemented a prototype of the Minion system in Java. Our implementation, designed to analyze concurrent Java libraries, utilizes the Soot bytecode instrumentation framework [51] for runtime monitoring, the WALA framework [2] for static analysis, and the Z3 theorem prover [10] for constraint solving.

Virtual threads. For ease of explanation, in our algorithms in Section 4, we considered each method invocation in the sequential test to originate from a distinct thread. In our implementation, we do not spawn these threads explicitly. Instead, we associate an invocation identifier with each method call, and treat these as "virtual" threads. Subsequently, when the Plan procedure needs to assign a thread to a target, we select the appropriate virtual thread. When necessary, a real thread is spawned in the client returned by CSolve.

Budget. In a given iteration, the number of threads in the client program can affect scalability. For example, if we use the solver-provided solution in an unconstrained manner in CSolve, then we might generate threads with low, if any, utility w.r.t. the client generation process. More significantly, such threads might contribute unnecessary constraints to the system. For instance, in Figure 3(c) there are no constraints on m3. A client that invokes this method with parameters b and 0 (parameters in the input test) from a new thread will not add new information.

To handle this, we encode an additional set of constraints, ϕ_{budget}, for CSolve. For each thread/invocation pair, we encode whether (or not) it is required to run in the next iteration. For threads that were previously assigned targets (by Plan), the encoding always evaluates to true. For other threads, whose usefulness is unclear, we let the constraint solver decide based on the given budget of allowed threads. Initially, we start with a budget of 3. We increment the budget in every subsequent iteration. The solution is used to determine the threads as well as the methods invoked.

Encoding null values. We also enable clients to invoke methods with null as a parameter value. Free variables for parameters (p_i) use legal object instances from a (past) client execution. Beyond that, we provide a special nil object, where the base pointer and all its fields point to null values. To enable assignment of the nil object to one of the fields of a free variable (e.g., y.f), we enforce that the base object (i.e., y) is not null.

Sub-goals, loops, native calls. The NextTgt procedure identifies relevant sub-goals to satisfy a conditional by performing a dynamic slice on the conditional. We track the flow of variables to the conditional, and derive the associated alias sets. Next, we detect updates to the elements of the sets. These updates are considered as potential sub-goals.

We handle loops via loop unrolling in the dynamic trace. We instrument all the library classes that are reachable from the class under test. Furthermore, collection classes are instrumented to detect modifications/crashes within the collection. If an uninstrumented class is used, then that class cannot be analyzed, which is a source of imprecision. To handle calls whose target is not available (as with native code, such as System.arraycopy for instance), we introduce synthetic models that are not necessarily executable but are accurate in modeling the semantics of the operation.

6. Experimental Validation

In this section, we report on our experimental evaluation of Minion and its comparison against other testing tools.

Experimental Setup and Benchmarks. We have applied Minion to several popular concurrent Java libraries. We analyzed the classes that are used in earlier work [43, 44], where the bugs lead to program crashes. We also focused in particular on well-tested classes within these libraries, all of which contain synchronization constructs and some of which are explicitly documented as thread safe. We conducted our experiments on an `Ubuntu-14.04` desktop machine with a 3.5Ghz Intel Core i7 processor with 16GB of RAM.

Table 2 provides information on the benchmarks. `java.lang`, `java.util` and `java.io` are packages from the Java Development Kit (JDK); `guava` is the Google Core Libraries for Java; `classpath` contains core libraries for VMs and compilers; `hsqldb` is a relational SQL database engine; and `cache4j` is a cache library for Java objects. For brevity we refer to the analyzed classes as `C1` through `C10`, as indicated in the table.

Benchmark	Version	Class name	|M|
cache4j	0.4	CacheCleaner (C1)	3
classpath	0.99	BufferedInputStream (C2)	10
guava	18.0	SimpleStatsCounter (C3)	6
hsqldb	2.3.3	DoubleIntIndex (C4)	32
java.lang	1.7	StringBuffer (C5)	50
java.io	1.8	CharArrayReader (C6)	8
		PipedReader (C7)	5
		PushbackReader (C8)	11
		StringReader (C9)	8
java.util	1.7	Vector (C10)	43

Table 2: Benchmark Information.

For each benchmark, we list in the table the version, class name and method count. We use version 1.7 for `C5` and `C10` due to soot compatibility issues. The key complexity factors in detecting concurrency-induced bugs are the number of methods, the number of parameters, as well as the number of conditionals along the path to the potentially buggy location. Given these factors, randomly invoking methods with arbitrary parameters concurrently will neither be scalable nor effective (as we also demonstrate experimentally below).

In our experiments, as the initial client, we have defined a program that randomly (and sequentially) invokes the methods in the class with random parameters. Each method is invoked exactly once. The targets include existing assertions and throw statements in the class, assertions on field dereferences and inserted assertions for previously known bugs. We set the budget for iterative refinement to five.

Performance Results. Table 3 presents the data pertaining to the detected concurrency-induced bugs and the internal analysis statistics. The number of targets for the classes ranges from 2 for `C1` to 23 for `C8`. Out of an overall of 80 targets, Minion was able to detect 31 *real* crashes due to concurrency-induced bugs, including previously unknown bugs. All of these bugs are true positives due to developer oversight.

As confirmation by a third party, we have already communicated some of the bugs to the respective developers, and are in the process of reporting all the other newly found bugs. We have already secured developer acknowledgement

Class	# of Targets	# of Constraints	Schedule length	Time (sec)	Violations	
					Seq.	Conc.
C1	2	254	37	21	0	1
C2	8	8189	25	70	2	6
C3	3	1898	43	10	0	2
C4	8	4105	10	94	4	4
C5	12	14646	116	664	5	1
C6	4	717	14	20	1	3
C7	10	3355	30	6	2	1
C8	23	7276	19	324	8	9
C9	4	900	20	21	1	3
C10	6	56045	74	120	0	1
Total	80			1350	23	**31**

Table 3: Detected bugs and analysis information.

for bugs reported on four of the benchmarks. The developers agreed that the reported scenario indeed represents an unexpected failure, and promised to take action in response. In one case, that includes changing the thread-safety documentation,[3] and in other cases, it involves fixing the code.[4]

Minion also triggered 23 crashes in a sequential context (i.e., crashes that manifest with only a single thread). These violations are mainly due to sanity checks on the input parameters (e.g., the input object is not null, etc) and are not bugs. These can be discarded using a switch exposed by Minion. For the remaining targets, which could not be reached using our system, we performed manual analysis of the code. While we cannot guarantee that our manual analysis is flawless, we could not — to the best of our ability — find a scenario that results in a real bug for any of these targets.

The third column in Table 3 indicates, for a given target, the average number of constraints discharged to the solver per iteration. The number of constraints is dependent on the code. Applying constraint solving to software applications faces major scalability challenges that need to be overcome (with millions of constraints generated for a single program as shown in [20]). As our approach is geared towards detecting bugs in library code, and the number of generated constraints is comparatively lower, scalability limitations due to the constraint solver become a lesser concern.

The average schedule length ranges from 10 for `C4` to 116 for `C5`. This corresponds to the ordering constraints on the instructions executed by the threads. For `C5`, the longer schedule is due to the presence of loop unrolling, apart from the complex schedule that needs to manifest for the bug to be detected. The overall time to execute the classes ranges from 6 seconds for `C7` to 11 minutes for `C5`. This is the time taken to either reach an empty stack of targets or generate a successful test. As our approach is catered towards finding complex bugs, we believe that the reported running times are acceptable.

[3] `https://github.com/google/guava/issues/2230`

[4] `https://bugs.openjdk.java.net/browse/JDK-8143394`

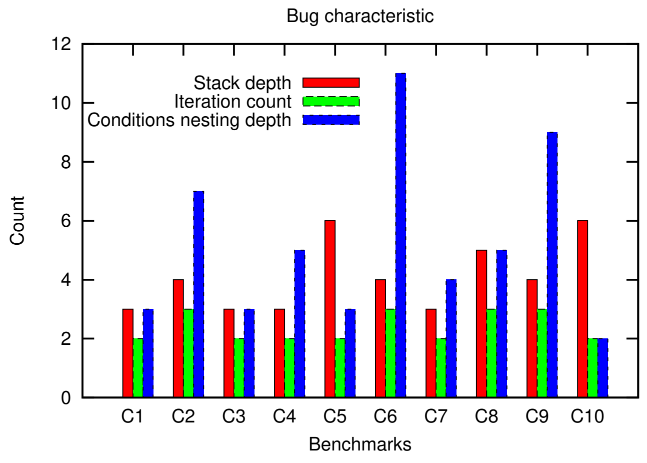


Figure 6: Characteristics of detected bugs.

Figure 6 provides insight into the complexity of the detected bugs. It presents data on the maximum values for stack depth, iterations, as well as nesting depth of the conditions (as specified by reach in the PLAN procedure). In C5, for instance, 6 methods are invoked in a nested manner between the client and the actual target. Similarly, for C6, the number of path conditions that *must* be satisfied to reach the target is 11. Here, five of the conditions are related to the input parameters, but the remaining conditions correspond to the implementation logic. Exposing the crash in a concurrent setting by satisfying these conditions is nontrivial. The iteration count to reach the target is 3, demonstrating the need to refine the clients to trigger the crashes.

Useability The output of MINION is a (multi-threaded) Java test program (with comments), leading to a crash, along with a human-readable schedule (in the form of a text document) to recreate the crash. When the client is run using MINION, the built-in scheduler consumes and follows the schedule, thereby inducing an exceptional/crashing run, making the problem visible/reproducible with no special effort.

Comparison with Other Tools. We also applied three existing approaches — CONTEGE [38], NARADA [44] and INTRUDER [43] — to the classes in our suite for comparative analysis. For this purpose, we used the publicly available implementation of these tools.

CONTEGE randomly generates many multithreaded tests (e.g., approximately 10K for C5); it was able to detect only two bugs across all classes in approximately eight hours. The first defect in C4 was detected relatively quickly (with 50 tests, and in less than five minutes). The second defect, in C10, took (much) longer to detect (around 4K tests and two hours). These results are not surprising due to the dependence of the approach on randomization.

The other two approaches required less than 10 minutes in total to analyze all the classes, and generated 48 multithreaded tests. Since these tests are geared towards detecting races and atomicity violations, and not crashes, we manually analyzed the possibility of the synthesized tests reaching the targets in the classes given *any* schedule. We observe that totally seven crashes could potentially be exposed by these two tools. The remaining 24 crashes reported by MINION could not be detected. This is because NARADA and INTRUDER are agnostic to path conditions and thread interleavings.

In summary, none of the three other tools was able to cause crashes beyond those caused by MINION. More importantly, MINION was able to detect many more crashes than those detected by all three of these tools combined.

```
37  public class PushbackReader extends FilterReader {
52    public PushbackReader(Reader in, int size) {
57      this.buf = new char[size];
58      this.pos = size;
59    }

71    private void ensureOpen() throws IOException {
72      if (buf == null)
73        throw new IOException("Stream closed");
74    }

106   public int read(char cbuf[], int off, int len)
                                     throws IOException {
107     synchronized (lock) {
108       ensureOpen();
109       try {
            ... // returns on len <= 0
118         int avail = buf.length - pos;
119         if (avail > 0) {
120          if (len < avail)
121           avail = len;
122          System.arraycopy(buf, pos, cbuf, off, avail);
             ...
126         }
            ...
135       } catch (ArrayIndexOutOfBoundsException e) {
136         throw new IndexOutOfBoundsException();
137       }
138     }
139   }

151   public void unread(int c) throws IOException {
152     synchronized (lock) {
153       ensureOpen();
154       if (pos == 0)
155         throw new IOException("Pushback buffer overflow");
156       buf[--pos] = (char) c;
157     }
158   }

247   public void close() throws IOException {
248     super.close();
249     buf = null;
250   }
281 }
```

Figure 7: Motivating example from openjdk-8u40-b25. Our report resulted in filing of Oracle JDK bug 8143394.

Case Study. We now discuss one of the failures exposed due to MINION that has been fixed by the developers. Figure 7 presents a code fragment from the latest version of the OpenJDK library [1]. In this code, there are several documented error conditions, such as the throwing of an IOException at line 155 backed by the throws statement at line 151. Beyond these situations, there are more subtle crash scenarios. We focus in particular on the arraycopy() call at line 122.

If buf is null, then a NullPointerException (NPE) would be thrown, which is unexpected based on the throws and catch statements in read().

Interestingly, this problem can manifest. It might even occur after a successful dereference of buf at line 118 within a synchronized context. This can occur due to a null assignment by another thread between the execution of the two lines. More specifically, there are two conditions, beyond invoking read, that need to be satisfied:

- unread() needs to be invoked to decrement pos (line 156), so that the condition at line 119 is satisfied; and
- close() needs to be invoked between lines 119 to 122 to set buf to null.

Apart from these conditions, the three method invocations should be on the *same* receiver object. While the NPE is also realizable at line 118, it is vital that the developer has a comprehensive understanding of the possible bugs (e.g., the exception at line 122) to introduce an appropriate fix.

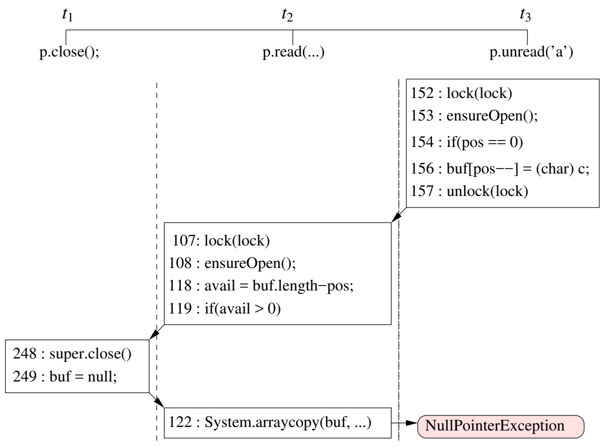


Figure 8: Synthesized failing concurrent execution.

Minion was able to synthesize the aforementioned complex failing scenario automatically by simply analyzing the execution of a sequential client that invoked the methods in the class with random parameter values. Figure 8 presents the generated multithreaded client and the corresponding failing execution.

7. Conclusions

In this paper, we described the design and implementation of a directed testing engine named Minion that combines dynamic trace information with statically identified refinement goals to iteratively refine a test, via constraint solving, to the point of exhibiting a concurrency-induced bug. Across 10 concurrent classes from popular Java libraries, Minion was able to detect 31 real crashes, some of which highly nontrivial, including previously unknown issues, in a total of 23 minutes. These results suggest that Minion is a compelling candidate for practical adoption.

Acknowledgements

We thank the anonymous reviewers for their constructive feedback that helped improve the presentation of this work. We also thank Peng Liu for useful initial discussions leading to this work. The first author is supported by a Google India PhD fellowship and the last author is supported by a Google Faculty Research award.

References

[1] PushbackReader.java. `http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/8u40-b25/java/io/PushbackReader.java/`.

[2] The Watson Libraries for Analysis. `http://wala.sourceforge.net/wiki/index.php/Main_Page`.

[3] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler. A few billion lines of code later: Using static analysis to find bugs in the real world. *Commun. ACM*, 53(2), 2010.

[4] S. Biswas, J. Huang, A. Sengupta, and M. D. Bond. Doublechecker: Efficient sound and precise atomicity checking. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 28–39, 2014.

[5] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, 2008.

[6] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. Exe: Automatically generating inputs of death. *ACM Trans. Inf. Syst. Secur.*, 12(2):10:1–10:38, Dec. 2008.

[7] S. Chandra, S. J. Fink, and M. Sridharan. Snugglebug: A powerful approach to weakest preconditions. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 363–374, 2009.

[8] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 258–269, 2002.

[9] C. Csallner, Y. Smaragdakis, and T. Xie. Dsd-crasher: A hybrid analysis tool for bug finding. *ACM Trans. Softw. Eng. Methodol.*, 17(2), May 2008.

[10] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, 2008.

[11] M. Eslamimehr and J. Palsberg. Sherlock: Scalable deadlock detection for concurrent programs. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 353–365, 2014.

[12] M. Eslamimehr and J. Palsberg. Race directed scheduling of concurrent programs. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 301–314, 2014.

[13] C. Flanagan and S. N. Freund. Fasttrack: Efficient and precise dynamic race detection. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, 2009.

[14] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 110–121, 2005.

[15] C. Flanagan, S. N. Freund, and J. Yi. Velodrome: A sound and complete dynamic atomicity checker for multithreaded programs. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, 2008.

[16] G. Fraser and A. Arcuri. Evosuite: Automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 416–419, 2011.

[17] P. Godefroid. Compositional dynamic test generation. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '07, pages 47–54, 2007.

[18] P. Godefroid, N. Klarlund, and K. Sen. Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, 2005.

[19] J. Huang. Stateless model checking concurrent programs with maximal causality reduction. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, pages 165–174, 2015.

[20] J. Huang, C. Zhang, and J. Dolby. Clap: Recording local executions to reproduce concurrency failures. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, 2013.

[21] J. Huang, P. O. Meredith, and G. Rosu. Maximal sound predictive race detection with control flow abstraction. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 337–348, 2014.

[22] V. Jagannath, M. Gligoric, D. Jin, Q. Luo, G. Rosu, and D. 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, 2011.

[23] P. Joshi, C.-S. Park, K. Sen, and M. Naik. A randomized dynamic program analysis technique for detecting real deadlocks. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, 2009.

[24] B. Kasikci, C. Zamfir, and G. Candea. Data races vs. data race bugs: telling the difference with portend. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2012, London, UK, March 3-7, 2012*, pages 185–198, 2012.

[25] P. Liu, X. Zhang, O. Tripp, and Y. Zheng. Light: Replay via tightly bounded recording. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, pages 55–64, 2015.

[26] B. Livshits, M. Sridharan, Y. Smaragdakis, O. Lhoták, J. N. Amaral, B.-Y. E. Chang, S. Z. Guyer, U. P. Khedker, A. Møller, and D. Vardoulakis. In defense of soundiness: A manifesto. *Commun. ACM*, 58(2):44–46, Jan. 2015.

[27] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, pages 329–339, 2008.

[28] N. Machado, B. Lucia, and L. Rodrigues. Concurrency debugging with differential schedule projections. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, pages 586–595, 2015.

[29] N. Machado, B. Lucia, and L. Rodrigues. Concurrency debugging with differential schedule projections. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, pages 586–595, 2015.

[30] N. Machado, B. Lucia, and L. E. T. Rodrigues. Production-guided concurrency debugging. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2016, Barcelona, Spain, March 12-16, 2016*, pages 29:1–29:12, 2016.

[31] S. McPeak, C.-H. Gros, and M. K. Ramanathan. Scalable and incremental software bug detection. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013.

[32] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997. ISBN 1-55860-320-4.

[33] M. Musuvathi and S. Qadeer. *Logic-Based Program Synthesis and Transformation: 16th International Symposium, LOPSTR 2006, Venice, Italy, July 12-14, 2006, Revised Selected Papers*, chapter CHESS: Systematic Stress Testing of Concurrent Software, pages 15–16. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[34] S. Nagarakatte, S. Burckhardt, M. M. Martin, and M. Musuvathi. Multicore acceleration of priority-based schedulers for concurrency bug detection. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, 2012.

[35] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder. Automatically classifying benign and harmful data races using replay analysis. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, 2007.

[36] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 75–84, 2007. ISBN 0-7695-2828-7.

[37] S. Park, S. Lu, and Y. Zhou. Ctrigger: Exposing atomicity violation bugs from their hiding places. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, 2009.

[38] M. Pradel and T. R. Gross. Fully automatic and precise detection of thread safety violations. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, 2012.

[39] D. Prountzos, R. Manevich, and K. Pingali. Synthesizing parallel graph programs via automated planning. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, pages 533–544, 2015.

[40] W. Pugh and N. Ayewah. Unit testing concurrent software. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 513–516, 2007.

[41] M. Samak and M. K. Ramanathan. Multithreaded test synthesis for deadlock detection. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '14, pages 473–489, 2014.

[42] M. Samak and M. K. Ramanathan. Trace driven dynamic deadlock detection and reproduction. In *Proceedings of the 2014 ACM SIGPLAN Conference on Principles and Practices of Parallel Programming*, PPoPP '14, 2014.

[43] M. Samak and M. K. Ramanathan. Synthesizing tests for detecting atomicity violations. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 131–142, 2015.

[44] M. Samak, M. K. Ramanathan, and S. Jagannathan. Synthesizing racy tests. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, pages 175–185, 2015.

[45] K. 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, 2008.

[46] Y. Smaragdakis, J. Evans, C. Sadowski, J. Yi, and C. Flanagan. Sound predictive race detection in polynomial time. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 387–400, 2012.

[47] S. Steenbuck and G. Fraser. Generating unit tests for concurrent classes. In *Proceedings of the 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, ICST '13, pages 144–153, 2013.

[48] B. Swarnendu, Z. Minjia, B. Michael, and L. Brandon. In *Proceedings of the 2015 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '15, 2015.

[49] S. Thummalapenta, T. Xie, N. Tillmann, J. de Halleux, and Z. Su. Synthesizing method sequences for high-coverage testing. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11.

[50] O. Tripp, O. Weisman, and L. Guy. Finding your way in the testing jungle: A learning approach to web security testing. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pages 347–357, 2013.

[51] R. Vallee-Rai, E. Gagnon, L. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing java bytecode using the soot framework: Is it feasible? In *In International Conference on Compiler Construction, LNCS 1781*, pages 18–34, 2000.

[52] C. Wang, S. Kundu, M. Ganai, and A. Gupta. Symbolic predictive analysis for concurrent programs. In *Proceedings of the 2Nd World Congress on Formal Methods*, FM '09, pages 256–272, 2009.

[53] C. Zamfir and G. Candea. Execution synthesis: a technique for automated software debugging. In *European Conference on Computer Systems, Proceedings of the 5th European conference on Computer systems, EuroSys 2010, Paris, France, April 13-16, 2010*, pages 321–334, 2010.